64'er

Neu überarbeitet:

SONDERHEFT ASSEMBLER

SONDERHEFT 35 ÖS 100,-/Sfr. 14,- Lit. 14000/hfl. 18,-/dkr. 72,- DM 14,-

Markt&Technik

64'er



Vollständige Kurse

- Verständlich erklärt: Assembler für Anfänger und Fortgeschrittene
- Der Schritt zum Profi: Maschinensprache perfekt beherrscht
- Mit vielen ausführlichen Beispielen

Komplette Programmier-Werkstatt

- Komfortable Assembler-Programmierung mit Hypra-Ass
- Reassembler – zurück zum übersichtlichen Code
- Speicher- und Diskettenmonitor der Spitzenklasse

Wichtige Tips & Tricks

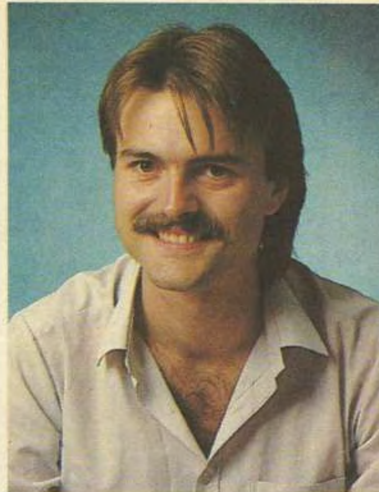
- Betriebssystem professionell genutzt

Assembler

Assembler

Alle Programme auch auf Diskette erhältlich





Die Macht des Lesers

Der Kontakt zum Leser ist für eine Redaktion, die eine gute Zeitschrift machen will, das A und O. Dies trifft in verstärktem Maße auf Computerzeitschriften und damit natürlich die 64'er-Sonderhefte zu.

So erreichten uns in den letzten Monaten gehäuft Anrufe und Briefe, deren Tenor lautete, daß das 64'er-Sonderheft 8/85 zum Thema Assembler nicht mehr zu erhalten ist und wie man es vielleicht doch noch bekommen könnte.

Gleichzeitig wurde vielfach der Wunsch an uns herangetragen, doch einen umfassenden Assembler-Kurs zu veröffentlichen, der einen Basic-Programmierer in den Rang eines Assembler-Experten erhebt.

Mit dem Ihnen hier vorliegenden Sonderheft haben wir beide Fliegen mit einer Klappe geschlagen und beide Wünsche erfüllt: Wir präsentieren Ihnen unter anderem den schon legendärenen Assembler-Kurs von Heimo Ponnath aus dem Sonderheft 8/85 in einer neu überarbeiteten Fassung. Dazu liefern wir selbstverständlich auch die nötigen Programmier-Tools wie den Assembler »Hypra-Ass«, den Maschinensprache-Monitor »SMON« und einen Reassembler, der Maschinencode wieder in lesbare Programme verwandelt.

Natürlich stand auch die Überlegung zur Debatte, einen völlig neuen Kurs zu schreiben. Doch warum das Rad noch einmal erfinden: Noch zu gut kann ich mich an den Sommer des Jahres 1985 erinnern, in welchem ich meine ersten Gehversuche in der schwierigen Materie Assembler unternahm. Nach mehreren Anläufen war es schließlich dieser Assembler-Kurs, der mir zum »Durchblick« verhalf. Und auch heute noch tut er als Nachschlagewerk gute Dienste.

Ebenfalls den Charakter eines Standardwerkes hat der Beitrag »Die Schritte zum Doktor der Maschinensprache«. Denn nur Übung macht den Meister und die alleinige Kenntnis der einzelnen Befehle des 6502-Prozessors macht noch keinen Star-Programmierer. Da muß schon die Erfahrung eines Profis her, die in diesem Beitrag auf leicht verständliche Weise an Sie weitergegeben wird:

Hier lernen Sie viele Programmiertricks, darunter Geschwindigkeitsoptimierung von Programmen und die effektive Ausnutzung der Routinen des C64-Betriebssystems.

Voraussetzung hierzu ist natürlich die Kenntnis, welche Routinen das Betriebssystem des C64 bereitstellt. Doch auch daran wurde gedacht:

Eine ausführliche Tabelle stellt die für den Programmierer wichtigsten Einsprungadressen und deren Funktion samt der benötigten Parameter vor.

Ihrem Einstieg in die Assemblerprogrammierung steht mit diesem Sonderheft also nichts mehr im Wege.

Ihr
Klaus Schrödl
(Redakteur)

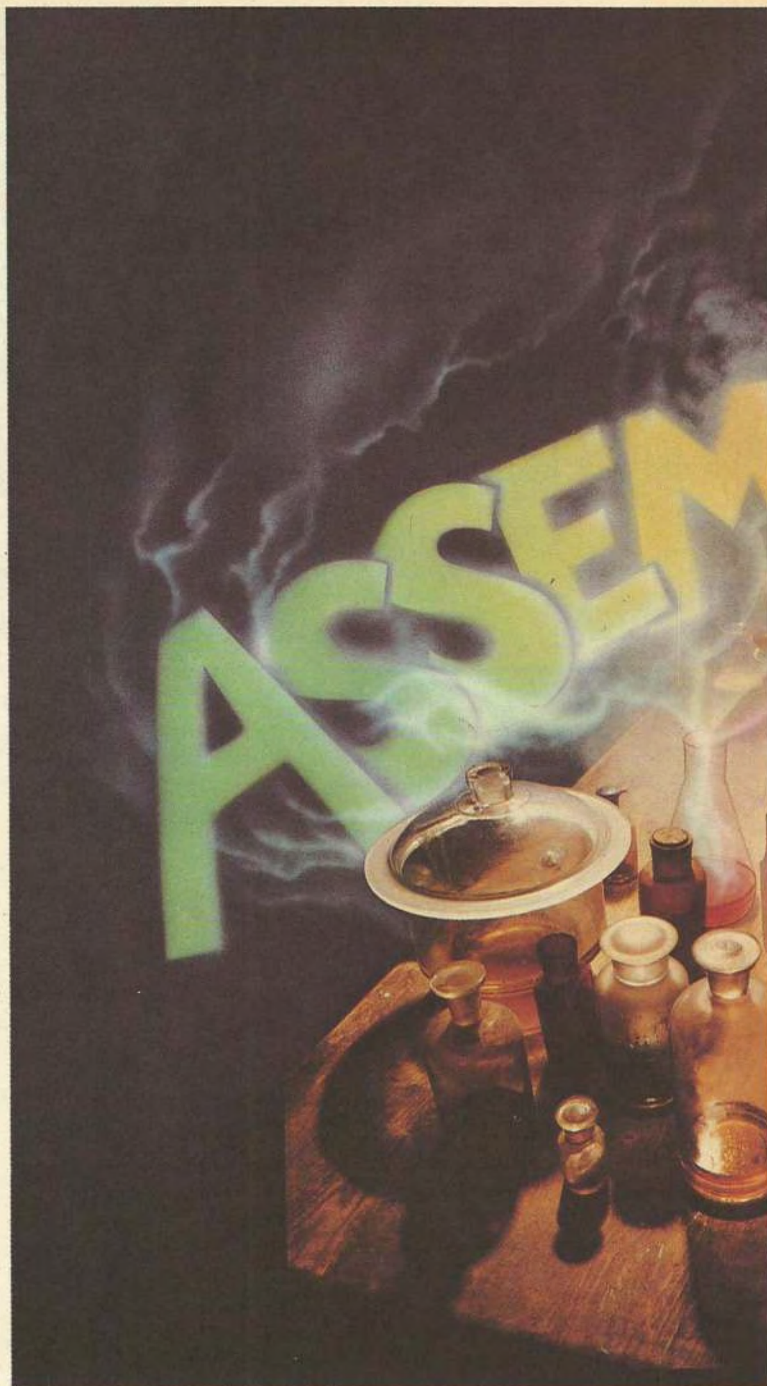
INHALT

Kurse



Eine komplette Werkstatt für die Assembler-Programmierung stellen wir Ihnen zur Verfügung. Tools für komfortable Programmierung, das Durchforschen von Software oder Aufspüren der Fehler sowie zur Rückwandlung von unübersichtlichem Maschinen-Code in lesbare Programmiersprache. **Seite 120**

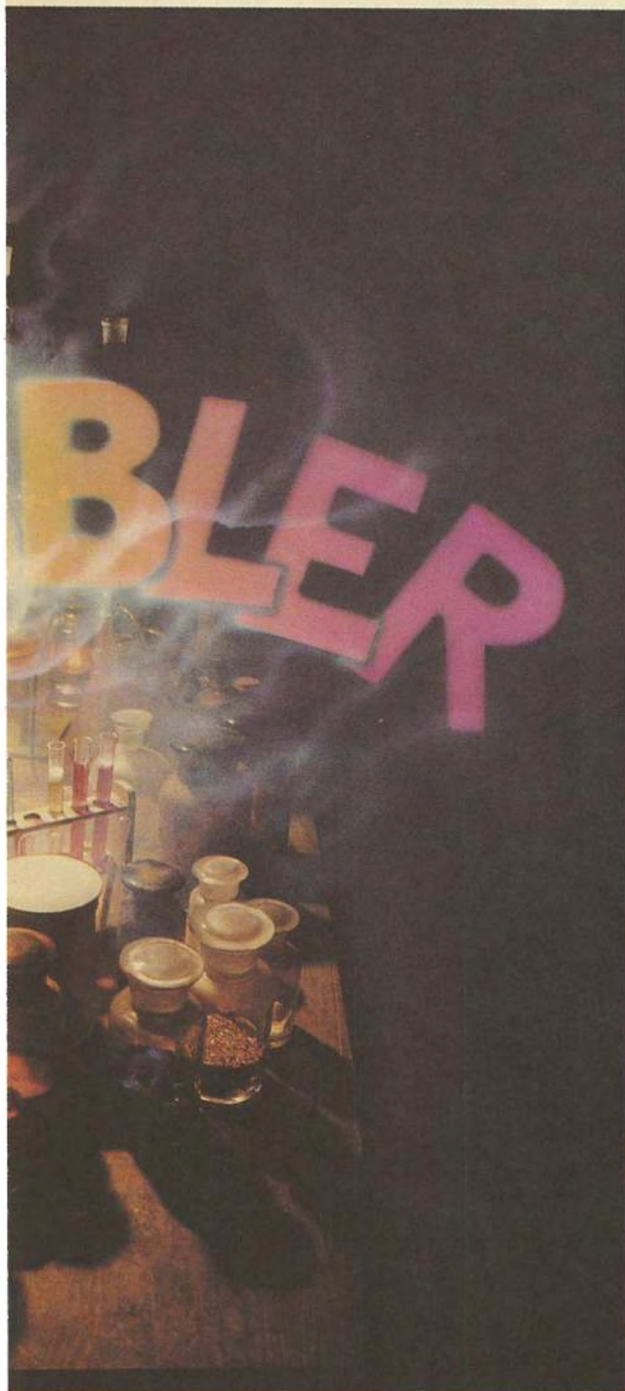
Tools



Den Prozessor des C64 direkt ansprechen – das erreichen Sie nennahe Programmierung eröffnet Ihnen Möglichkeiten, Der Kurs begleitet Sie von den Anfängen bis zum fortge-

Tabellen

Tips & Tricks



Tricks Tips

Sie brauchen das Rad
nicht noch einmal erfinden.
Viele Assembler-Routinen sind bereits im Betriebssystem des C64 enthalten. Welche wichtigen Routinen gibt es und wie kann man sie nutzen?
Hier die Lösungen. **Seite 158**

Den Weg zum »Dr. Assembler« ebnet Ihnen der zweite Kurs. Ein Profi verrät all seine Tricks, damit Sie noch effektivere Software entwickeln können.
Seite 90



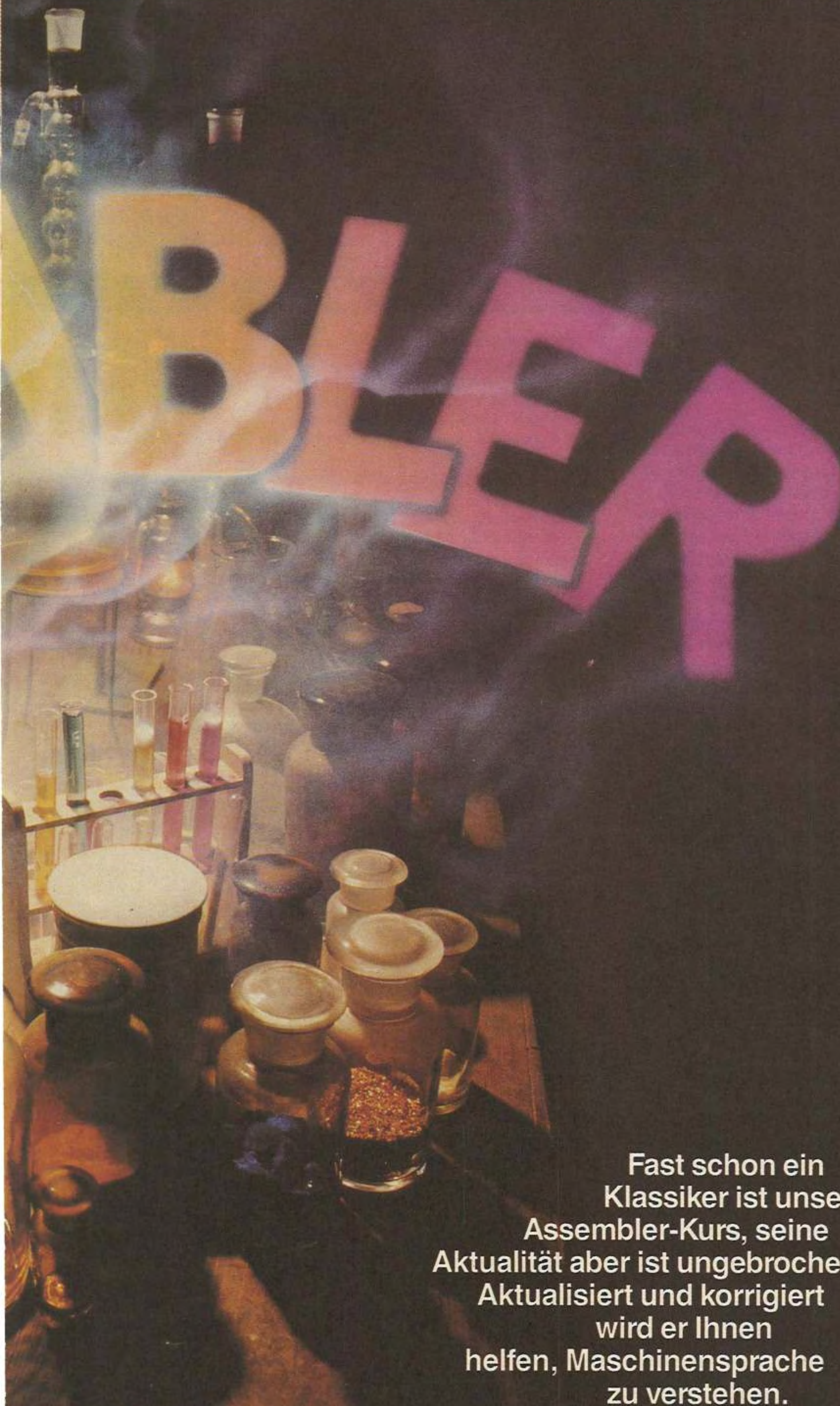
...mit der Programmiersprache »Assembler«. Die maschi-
...die weit über das in Basic Machbare hinausgehen.
...schrittenen Assembler-Programmierer. **Seite 6**

Alle Programme aus Artikeln mit einem ▪ -Symbol finden Sie auch auf der Programmservice-Diskette zu diesem Sonderheft

Vermutlich hat es Ihnen auch schon ab und zu in den Fingern gejuckt, wenn Sie von Wunderdingen gelesen haben, die man per Maschinensprache mit dem Computer programmieren kann. Vielleicht haben Sie sogar schon mal nichtsahnend angefangen einzutippen, was Sie als Assemblerlisting sahen. Doch schon nach dem ersten »C000 LDA # $00« und RETURN weigerte sich der Computer mit einem lapidaren »SYNTAX ERROR«. Wieso, werden Sie sich gefragt haben, das ist doch nun die Sprache unserer Maschine, nämlich Maschinensprache, was habe ich falsch gemacht?

Dann sind Sie sicherlich mal auf diese merkwürdigen Basic-Programme gestoßen, in denen ein langer Wurm von DATA-Zeilen mit einem kleinen FOR..NEXT.. POKE-Kopf vorne und einem SYS-Schwanz hinten enthalten ist, und die man Basic-Lader nennt. Sie haben fleißig Zahlen eingetippt - das Ganze hoffentlich sofort gespeichert -, vorschriftsmäßig mit dem SYS-Befehl gestartet und auf einen scheintoten Computer geschaut, der nur noch durch Aus- und Einschalten wiederzubeleben war. Wenn Sie dann nach langer Fehlersuche den irrtümlich eingetippten Punkt durch ein Komma ersetzt haben (oft finden Sie auch keinen Fehler, denn bei langen DATA-Sequenzen schlägt der Druckfehlerteufel mit Vorliebe zu), werden Sie sich gefragt haben, warum in aller Welt dieses kleine Mißgeschick den ganzen Computer abstürzen läßt. Sie merken vermutlich schon, daß mir das alles und noch mehr (worüber ich schamhaft schweige) passiert ist. Die Konsequenz war, daß ich losging, um ein schlaues Buch zu erwerben. Aber merkwürdig, damals tauchte der Begriff »Maschinensprache« in keinem Titel auf. Irgendwann begriff ich, daß Assembler und Maschinensprache irgend etwas miteinander zu tun haben.

ASSEM… ist keine



Fast schon ein Klassiker ist unser Assembler-Kurs, seine Aktualität aber ist ungebrochen, Aktualisiert und korrigiert wird er Ihnen helfen, Maschinensprache zu verstehen.

Alchimie

Aber da fing das ganze Elend erst richtig an: Da gab es 6502-, Z80-, 8080-, 8085-, 6800-Assembler, da waren irgendwelche Schaltpläne, anscheinend, wie man wo was hinlötet –, für mich als Nichtelektroniker eine Art moderner Kunst –, da war von CPU, Bussen, negativen Flanken, Zweiphasentakten die Rede.

Ich habe mich furchtbar geärgert über die Geheimsprache, die es dem Uneingeweihten verwehrt, etwas zu verstehen. Seither hat sich einiges verändert. Die Geheimnisse sind keine mehr und ich werde Ihnen in dieser Serie ohne verschlüsselte Sprache die magischen Zirkel der Assembler-Alchimisten offenbaren. Heute gibt es auch Bücher über »Maschinensprache auf dem Commodore 64« und es sei Ihnen angeraten, ruhig auch das eine oder andere durchzuarbeiten. Sie werden allerdings feststellen, daß die meisten davon gerade dort aufhören, wo es anfängt spannend zu werden: bei der Benutzung von Routinen des Betriebssystems und des Interpreters. Deswegen soll der Schwerpunkt dieses Artikels woanders liegen:

Wir werden das notwendige Grundwissen über die Hardware nur ganz knapp behandeln, dann das Vokabular des 65xx-Assemblers kennenlernen. Den Hauptteil des Kurses verbringen wir aber mit Dingen, über die es kaum Literatur gibt, nämlich wie man für eine Unzahl von Programmieraufgaben nicht nochmal das Rad erfinden muß, weil es schon längst in unserem Computer existiert.

Bevor wir loslegen, will ich Ihnen noch etwas Literatur empfehlen:

a) Wenn wir über den Speicheraufbau, das binäre und das hexadezimale Zahlensystem reden, können Sie mehr dazu im Buch »C 64: Wunderland der Grafik« (Markt & Technik Verlag, Haar) nachlesen. Besonders wichtig sind die Kapitel 1.2 und 2.

b) Als Nachschlagebuch sehr wertvoll ist das Buch von

Raeto West: C 64 Computer Handbuch. Hier finden Sie auch viele Tips und Tricks.

c) Später wird Ihnen dieses Buch fast unentbehrlich vorkommen: R. Babel, M. Krause, A. Dripke: Systemhandbuch zum Commodore 64 (und VC 20), München 1983

Weitere Literaturempfehlungen werde ich Ihnen von Fall zu Fall geben und Sie finden sie auch von Zeit zu Zeit in der Bücherecke des 64'er-Magazins. Gerade zu unserem Computer erscheint fast jeden Monat ein neues Buch und es ist nicht einfach, die Spreu vom Weizen zu trennen.

1. Einige Begriffserklärungen

Zunächst einmal muß ich Sie enttäuschen: Ich glaube kaum, daß Sie mit Ihrem Computer je einmal in Maschinensprache verkehren werden!

Maschinensprache, das ist die einzige, die der Computer direkt versteht, das sind vorhandene oder nicht vorhandene Stromimpulse oder Magnetisierungszustände, die bei unserem Computer durch 8-Bit-Binärzahlen auszudrücken sind. Was wir mit unserem Computer reden werden ist Assembler. Mit dem Computer sprechen soll heißen: Mit dem Gehirn unseres Computers, dem Prozessor, oft auch CPU (von Central Processing Unit=Zentraler Arbeitsbaustein) genannt, verkehren, also ihm Befehle zu geben. Solche CPUs werden bei verschiedenen Firmen hergestellt, sind daher unterschiedlich aufgebaut und auch unterschiedlich ansprechbar. Ein weit verbreiteter Prozessortyp ist der 6502, der das Gehirn des C 64 und auch des C 128 (Dort heißt er dann 8502) ist. Genau genommen ist das Gehirn des C 64 allerdings der 6510, ein dem 6502 fast identischer Prozessor. Auf die kleinen Unterschiede werden wir noch zu sprechen kommen.

Beide (8502 und 6510) sind in 6502-Assembler zu programmieren und wenn wir diese Sprache sprechen, sind für uns alle 6502-Computer zugänglich: Commodore, Apple II, Atari XL/XE und einige andere.

Nun wissen Sie aber immer noch nicht, was Assembler eigentlich ist. Das englische Wort »assemble« heißt auf deutsch etwa montieren, zusammenstellen. Es handelt sich also um eine Programmiersprache und weil sie sehr eng am Computer orientiert ist, spricht man von einer »maschinenorientierten« Programmsprache im Gegensatz zu »problemorientierten« Programmsprachen wie Basic, Pascal, Comal etc., die - so sollte es jedenfalls sein - auf jedem Computertyp gleich aussehen.

Ein Assembler ist aber noch etwas anderes, nämlich ein Software-Instrument, das einen in Assembler geschriebenen Befehl in Maschinensprache übersetzt. Man spricht vom Vorgang des Assemblierens. Das umgekehrte leistet ein Disassembler, welcher uns Maschinensprache durch Rückübersetzung lesen hilft. Um die Verwirrung noch zu steigern, sage ich Ihnen auch noch, was ein Monitor ist. In diesem Zusammenhang ist kein Bildschirmgerät damit gemeint, sondern ein Software-Instrument, das Einblick in die Register und Speicher des Computers gewährt.

2. Basic contra Assembler

Damit Sie nun den Überblick völlig verlieren, sei abschließend zu diesem Sprachenwirrwarr noch erzählt, daß Software-Pakete, die sowohl Assembler als auch Disassembler als auch Monitor enthalten und noch eine Menge anderer brauchbarer Dinge, oft als »Assembler« angeboten werden. Das ist ein alter Trick der Alchimisten, verschiedenen Dingen den gleichen Namen zu geben!

Um das Nachfolgende deutlich zu machen, schalten Sie bitte Ihren Computer an und tippen die beiden folgenden Programme ein, die beide genau dasselbe tun: Das obere Viertel unseres Bildschirmes mit dem Buchstaben A. Zunächst einmal in Basic:

```
10 FOR I=1024+255 TO 1024 STEP-1
20 POKE I, 1:POKE I+54272,7
30 NEXT I
```

Das Programm braucht 55 Byte + 7 Byte für die Variable I, macht zusammen 62 Byte Speicherplatz. Es geht ganz schnell und wenn Sie es schaffen, können Sie ja mal mitstoppen, wie lange es von RUN bis READY braucht: zirka 4 Sekunden.

Jetzt dasselbe in Assembler. Weil wir aber noch nicht soweit sind, erst mal als Basic-Lader, der uns das Programm in den Speicher bringt (wir kommen dazu gleich noch). Geben Sie also NEW ein und dann:

```
10 FOR I=7000 TO 7000+16
20 READ A:POKE I,A:NEXT I :END
30 DATA 160,255,162,7,169,1,153,255,
3,138,153,255,215,136,208,244,96
```

Starten Sie den Basic-Lader mit RUN und nach dem READY geben Sie NEW und CLR ein: wir brauchen ihn nicht mehr. Ab Speicherstelle 7000 steht jetzt unser Assemblerprogramm als Maschinencode. Daß es wirklich dasselbe tut wie das Basic-Programm erfahren Sie durch SYS 7000. Da hatten Sie vermutlich gar keine Zeit mehr, auf die Stoppuhr zu drücken! (5,4 Millisekunden etwa dauert das ohne die Zeit, die der Basic-Interpreter für den Befehl SYS benötigt.) Außerdem braucht das Programm 17 Byte Speicherplatz.

INH[ALT]

Genau das ist es, was die Assemblerprogrammierung so reizvoll macht: Der Speicher faßt mehr an Programm und die Ausführung des Programmes geht fast 1000mal so schnell! Dazu kommen natürlich noch einige andere Kriterien, denn viele Probleme sind zum Beispiel in Basic nicht lösbar, sondern nur mit dem vielseitigeren Assembler.

Unser Computer ist darauf vorbereitet, daß wir ihn in Basic ansprechen. Er enthält im Normalfall sofort nach dem Einschalten ein stets präsentes Übersetzungsprogramm, den Interpreter, welcher unsere Basic-Anweisungen für ihn verständlich interpretiert. Auch das ist ein Unterschied zu Assembler-Programmen: Ist ein solches Programm erst einmal assembliert (also als Maschinensprache im Speicher vorhanden), braucht man kein Übersetzungsprogramm mehr. Basic-Programme dagegen müssen bei jedem Durchlauf von vorne bis hinten ständig übersetzt werden, sie laufen nicht ohne vorhandenen Interpreter. Wie so ein Interpreter im Prinzip arbeitet und was ihn von einem sogenannten Compiler unterscheidet, können Sie im 64'er Sonderheft 6 (Top-Themen) im Artikel von M. Törk über seinen Strubs-Precompiler nachlesen.

Dort sehen Sie dann auch, daß ein Compiler zwar ein Basic-Programm enorm beschleunigen kann, aber bei weitem nicht an die Geschwindigkeit reiner Assemblerprogramme heranreicht, vom Speicherplatzbedarf ganz zu schweigen.

Leider hat der C 64 keinen eigenen Assembler implementiert. (Sie merken, daß jetzt von dem Software-Paket die Rede ist!) Es gibt einen etwas mühseligen Weg, dieses Handicap zu umgehen: den Basic-Lader. Vielleicht haben Sie schon einmal ein solches Programm abgetippt. Wie ist

A L T

32. Die Urzelle eines Programmprojektes
33. Wir stapeln
34. Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS
35. Sein oder Nichtsein: das Rätsel des Prozessorports
36. Die indirekte Adressierung
37. Die ersten Kernel-Routinen
38. Der C 64 und Fließkommazahlen
39. Die beiden ersten Interpreter-Routinen
40. Assembler-Befehle zum Beherrschen von Bits
41. Die restlichen Bit-Verschiebe-Operationen
42. Schneller Joystick
43. Die 16-Bit-Multiplikation
44. 16-Bit-Division
45. Das Programmprojekt wird fortgeführt
46. Die ROM-Bereiche als Datenquellen
47. Was sind Interrupts?
48. Das Unterbrechungssystem der CPU 6510/6502
49. Schlüssel zur Unterbrechungsprogrammierung: CLI, SEI, RTI, BRK
50. Woher kommen die Unterbrechungsanforderungen?
51. Der VIC-II-Chip als Unterbrechungsquelle
52. Die beiden CIA-Bausteine als Unterbrechungsquellen
53. Der IRQ-CIA
54. Der NMI-CIA
55. Die Restore-Taste und ein kleines Testprogramm
56. Der normale Verlauf eines IRQ
57. BRK-Unterbrechung
58. Was macht ein NMI?
59. Eigentlich keine Unterbrechung: Reset
60. Die Sache mit dem Modulstart
61. Nutzung der Unterbrechungen
62. Ein Programm zum VIC-II-IRQ
63. Unterbrechungen mit den CIAs
64. Die Timer der CIAs
65. Die Echtzeituhren

3. Wie sag ich's meinem Computer?

also der Weg, mit einem solchen Lader eigene Maschinenprogramme in den Computer zu bekommen?

a) Erstellen des Assembler-Programmes. Das zu lernen ist die Hauptaufgabe in diesem Artikel. Das Ergebnis wird eine Kette von Befehlen sein, zu denen zum Beispiel der Befehl RTS gehört.

b) Jedem Befehl in Assembler entspricht in Maschinensprache ein Binärcode in einer Speicherstelle. Diese Codes sind in Listen nachschlagbar: RTS entspricht beispielsweise dem Binärcode 0110 0000.

c) Der Code muß in eine Speicherstelle eingegeben werden. Das geschieht von Basic aus mit dem POKE-Befehl. Weil aber Basic keine Binärzahlen kennt, muß der Code ins Dezimalsystem umgerechnet werden. Glücklicherweise sind in den Tabellen meist schon die Codes als Dezimal- oder wenigstens als Hexadezimalzahlen enthalten. RTS ist dezimal 96 (oder hexadezimal 60, das auch $ 60 geschrieben werden kann). Man POKEt nun an die richtige Adresse den Wert 96, also zum Beispiel POKE 7016,96

d) Auf diese Weise wird Byte für Byte in der Programmabfolge verfahren. Das reine POKEn geschieht dann eben in der Form wie im oben gezeigten Basic-Lader. Mühsam, mühsam! Auch kann man leider nur mit dem PEEK-Kommando nachsehen, was denn nun im Speicher steht (PEEK (7016) gibt uns den Wert 96, entsprechend RTS).

Ein anderer Weg ist, den in diesem Sonderheft auf Seite 132 abgedruckten »SMON« abzutippen, oder sich die Leser-Service-Diskette zu bestellen.

Assembler (das Software-Paket) gibt es in den unterschiedlichsten Ausführungen. Es gibt beispielsweise Direkt-Assembler, die jede Programmzeile sofort nach dem RETURN assemblieren, aber auch 2-Pass-Assembler, bei denen das erst nach Abschluß des Programms insgesamt durch einen Befehl (zum Beispiel ASSEMBLE) geschieht. Bei einigen kann man (ähnlich wie bei Basic mit REM) Kommentare anfügen, bestimmten Programmstellen Namen geben (LABEL), ganze Programmabschnitte mit einem Merknamen aufrufen (MAKROS) und so weiter. Was Sie für sich bevorzugen, bleibt Ihnen natürlich überlassen. Die in diesem Artikel beschriebenen Programme werden am Anfang auf diese schönen Erleichterungen verzichten, es wird sozusagen der nackte Assembler verwendet. Was Sie aber außer dem reinen Assembler noch brauchen, ist ein Disassembler und ein Monitor (ich habe schon erklärt, welchen ich meine), damit wir unseren Computer (fast) immer im Griff haben. Alle diese Werkzeuge eines Assembler-Profis finden Sie in diesem Sonderheft.

4. Wie funktioniert unser Computer?

Weil das Programmieren in Assembler einen viel engeren Kontakt zu technischen Einzelheiten unseres Computers erfordert, ist es notwendig, ein wenig über diese Innereien und ihre Funktion zu wissen. Sehen Sie sich dazu bitte das Bild 1 auf Seite 12 an.

Da sehen wir zunächst unseren Mikroprozessor, der meist eine Menge Funktionen in sich vereinigt (dazu kommen wir noch). Im Prinzip ist das unsere CPU (Zentraler Arbeitsbaustein). Der Prozessor steht über eine Reihe von Leitungen mit dem Rest des Computers in Verbindung. Diese Leitungen werden im Fachjargon BUSSE genannt. Da ist zunächst einmal der sogenannte Adreßbus, auf dem 16-Bit-Adressen transportiert werden, die der Prozessor erzeugt, und die die Herkunft oder auch das Ziel von Daten festlegen, die über den Datenbus laufen. Dieser kann 8-Bit-

Markt&Technik

64'er SOFTWARE EXTRA









64'er Extra: Grafik Vol. 1

Giga-CAD. Unschlagbare 3-D-Konstruktion auf dem C64. Hi-Eddi. Das Super-Zeichen- und -Malprogramm. Title Wizard. Giga-CAD-Filme für eigene Vorspänne. Pic-Loader. Verwenden Sie Hi-Eddi-Grafiken für eigene Programme. Hi-Maus. Maus-Treiber für Hi-Eddi. Hi-Spiegel. Spiegeln Sie beliebige Ausschnitte einer Grafik. Filmconverter. Giga-CAD-Filme können mit diesem Programm in das Hi-Eddi-Format umgewandelt werden. Druckeranpassungen für Hi-Eddi: Printer/Plotter VC 1520, MPS-801/802/803, Seikosha GP 700VC, Star NL-10, C.Itoh-8510, C.Itoh Riteman C+.

Bestell-Nr. 38701
DM 49,90* (sFr 44,90*/öS 499,-*)

64'er Extra: Grafik Vol. 2
Bestell-Nr. 38702
DM 39,90* (sFr 34,90*/öS 399,-*)

64'er Extra: Grafik Vol. 3
Bestell-Nr. 38703
DM 39,90* (sFr 34,90*/öS 399,-*)

Adventure-Pack Vol. 1

Robox: Fesselndes Grafik-Science-Fiction-Adventure. Der Herrscher eines fremden Planeten ließ sein Gehirn nach seinem Tod künstlich weiterleben – in einem Körper ohne Seele. Ihre Aufgabe ist es, zu Robox zu gelangen und ihn unschädlich zu machen, um die Erde vor dem sicheren Tod zu bewahren. Wie Sie es tun, bleibt Ihnen überlassen. Mit dem mitgelieferten Fall-Editor konstruieren Sie weitere Verbrechen und geben damit ihren Freunden harte Nüsse zu knacken. **Scotland Yard:** Spannendes Kriminal-Adventure. Begeben Sie sich auf spannende Verbrecherjagd in das London des 19. Jahrhunderts und lassen Sie sich bei Scotland Yard engagieren.
Mit dem mitgelieferten Fall-Editor konstruieren Sie weitere Verbrechen und geben damit Ihren Freunden harte Nüsse zu knacken.

Bestell-Nr. 38704
DM 29,90* (sFr 24,90*/öS 299,-*)

64'er Extra: Disk-Utilities Vol. 2

Disk-Mon 64: Professioneller Floppy- und Diskettenmonitor. Master-Copy: Backup-Kopierprogramm. Dual-Filecopy: File-Kopierprogramm für zwei Laufwerke. Track-Copy: Einfaches Kopieren und Formatieren von einzelnen Tracks. Tornado-Copy 1571: Schnelles Backup-Programm für einseitig bespielte Disketten. Hypra-Load/Save: Software-Speeder für C64. Hypra-Perfekt: Hypra-LOAD/SAVE, eingebunden ins Betriebssystem. EXOX V3: Leistungsfähiges Betriebssystem für C64. ProDisk: Komfortable Diskettenverwaltung in Assembler. EX.DIR & BAM: Ausführliches Directory. Hypra-Format 1541: Formatieren einer Diskette in nur 8 Sekunden. Disk-Searcher: Findet sehr schnell beliebige Zeichenketten auf Diskette. File-Manager: Befehlserweiterung zur Verwaltung von Disketten.

Bestell-Nr. 38707
DM 49,–* (sFr 44,-*/öS 490,-*)

64'er Extra: Disk-Utilities Vol. 1
Bestell-Nr. 38706
DM 49,–* (sFr 44,-*/öS 490,-*)

64'er Extra: Programmier-Utilities Vol. 1

Diese Sammlung leistungsfähiger Basic-Befehlserweiterungen ermöglicht es Ihnen, mit geringem Aufwand hochwertige Programme zu schreiben. **Hypra-Basic:** Mit dieser modular aufgebauten Befehlserweiterung wird es Ihnen möglich, je nach Anwendungsgebiet Befehle und Funktionen zusammenstellen. **Special-Basic:** Über 200 neue Basic-Befehle, die unter anderem die Bereiche Programmeditor, strukturierte Programmierung, komfortabler Zeichensatz, Sound sowie Ein-/Ausgabe- und Diskettenzugriffe umfassen, helfen Ihnen in fast allen Situationen, schnell und effektiv zu programmieren.

Bestell-Nr. 38716
DM 39,–* (sFr 35,-*/öS 390,-*)



Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0.
Bestellungen im Ausland bitte an: SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656,
ÖSTERREICH: Markt&Technik Verlag Gesellschaft m.b.H., Große Neugasse 28, A-1040 Wien, Telefon (0222) 5871393-0,
Rudolf Lechner&Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526,
Ueberreuter Media Verlagsges.mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (222) 481543-0

232/806

SOFTWARE DER EXTRAKLASSE



64'er Extra: C16 – C116 – Plus/4 MasterBase

Das Programm MasterBase bietet unter anderem folgende Möglichkeiten: Benutzerführung durch Pull-down-Menüs und Windows, maskierte Suchmöglichkeiten, indexsequentieller Zugriff, Reorganisation von zerstörten Datenbeständen, komfortabler Editor, u. a. zur Erstellung von Datei-Masken, Feldattributen etc., optimale Druckeranpassung, feldspezifische ESC-Sequenzen, Parameterdateien für seriellen und parallelen Druckerbetrieb, vorgefertigte und erweiterbare Code-Tabellen, Tastatur-Makros, Daten-Im- und -Export, Code-Wandlung von externen Dateien, Erstellung von Serienbriefen oder Rundschreiben.

Bestell-Nr. 38719

DM 49,–* (sFr 44,–*/öS 490,–*)



64'er Extra: Abenteuer-Spiele

Sein letzter Trick. Chicago zur Zeit der Prohibition: Beim Kartenspielen gewinnen Sie eine kleine Brennerei. Kurz vor der Ausführung eines großen Auftrags fliegt Ihre Brennerei auf. Bei einem Boß aus der Mitte Chicagos, Don Spazzatura, erstehen Sie »Ersatzmaterial«. Leider werden Sie von Spazzatura betrogen, er hat Ihnen nur Wasser verkauft! Jetzt schwören Sie sich nur eines: Rache für Don Spazzatura.

Wanderung. Irgendwann in ferner Zukunft: Sie sind der einzige Überlebende eines Raumschiffabsturzes. Ihre einzige Überlebenschance besteht darin, den nächsten Raumhafen zu erreichen – aber Sie müssen sich beeilen, denn Ihr Sauerstoffvorrat ist begrenzt....

Bestell-Nr. 38715

DM 39,–* (sFr 35,–*/öS 390,–*)



128'er Extra: The Best of 128'er

Hier finden Sie die besten Programme für den C 128, die im 64'er Magazin und in den Sonderheften veröffentlicht werden. MASTERTEXT: Professionelle Textverarbeitung. COLOR-PACK 1: Super-Grafikerweiterung (480 · 240 Punkte Auflösung). TOP-FLOP: Leistungsfähiger Diskettenmonitor. DOUBLE-ASS: Zwei-Paß-Assembler, Unterstützung des Z80. WINDOW-TECH: Betriebssystem-Erweiterung, Unterstützung von 10 Windows. CENTRONICS-SCHNITTSTELLE: Unterstützung beliebiger CENTRONICS-Drucker. MICRO-HARDCOPY: Gestochen scharfe Hardcopys für Epson-Drucker und Kompatible. VECTORS: Super-Spiel im 80-Zeichen-Modus. UNIBOOT: Bootsektor manipulieren.

Bestell-Nr. 38712

DM 49,–* (sFr 44,–*/öS 490,–*)



128'er Extra: Paint R.O.I.A.L.

Paint R.O.I.A.L ist eines der wenigen Malprogramme, die die höchste Auflösung Ihres C128 verwenden. Wahlweise können sogar alle 16 Farben verwendet werden. Leistungsmerkmale: Auflösung: 640 · 200 Punkte (schwarzweiß) 640 · 176 Punkte (Farbe), vielfältige Blockoperationen: Kopieren, Löschen, Laden, Speichern, Spiegeln, Rotieren, beliebiges Vergrößern und Verkleinern, wahlweise Ausführung aller Zeichenfunktionen mit Pinsel oder Sprühdose, definieren von Grafikfenstern, leistungsfähige Pinselfunktion mit frei definierbaren Pinselformen, Sprühdosenfunktion, kombinierbar mit den zwölf Pinselformen und Mustern, Radiergummi, UnDo-Funktion, Übernahme von C64-Bildern, Laden aus dem Directory.

Bestell-Nr. 38736

DM 49,–* (sFr 44,–*/öS 490,–*)

*Unverbindliche Preisempfehlung



Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computer-Fachgeschäften oder bei Ihrem Buchhändler

Fragen Sie Ihren Fachhändler nach unserem kostenlosen Gesamtverzeichnis mit über 500 aktuellen Computerbüchern und Software. Oder fordern Sie es direkt beim Verlag an!

Daten transportieren, und zwar schreibend oder lesend, also zum Beispiel vom Prozessor zum RAM (schreibend), vom RAM zum Prozessor (lesend) und so weiter. Außerdem gibt es da noch einen Steuerbus, der verschiedene Synchronisationsaufgaben durchführen hilft. Links vom Prozessor ist ein Taktgeber angedeutet. Damit nichts durcheinander kommt, läuft alles im Computer sozusagen im Gleichschritt. Diese Uhr ist gewissermaßen der Trommler, den Sie vielleicht von den alten Ruder-Galeeren kennen.

und herschieben wollen. Sowohl der Akku (so werde ich ihn, in der Hoffnung auf Ihr wohlwollendes Verständnis, künftig bezeichnen) als auch alle anderen Register (mit einer Ausnahme) sind 8-Bit-Register, das heißt, die höchste Zahl, die darin bearbeitet werden kann, ist 255 (binär 1111 1111). Nahezu ebensooft wie den Akku werden wir die beiden sogenannten Index-Register X und Y benutzen. Warum man sie Index-Register nennt, werden Sie noch im Verlauf des Kurses sehen. Als nächstes zum Prozessor-

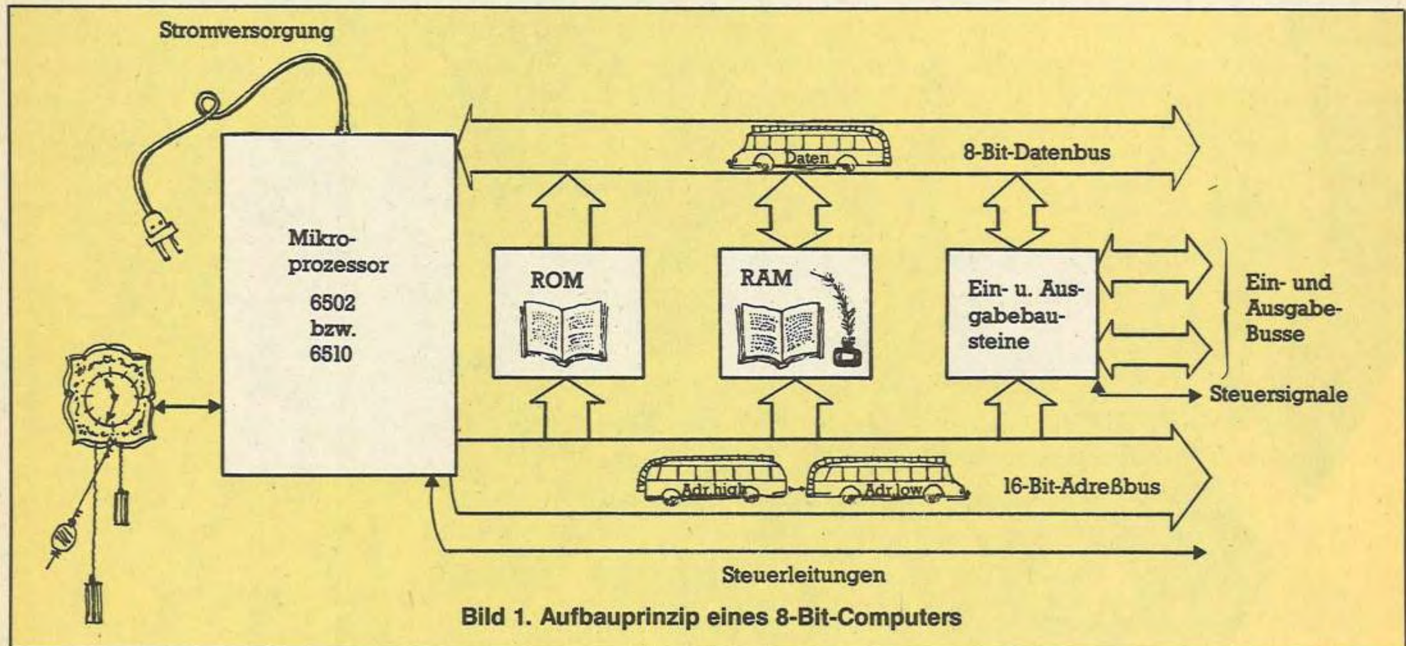


Bild 1. Aufbauprinzip eines 8-Bit-Computers

Dann sehen Sie einen ROM-Bereich, also einen Nur-Lese-Speicher (Read Only Memory). Daß man hier nur herauslesen kann, ist durch den Pfeil zum Datenbus gekennzeichnet. Doppelpfeile finden wir aber beim RAM (Random Access Memory), einem Speicher für beliebigen Zugriff, also lesend und schreibend, und bei den Ein- und Ausgabebausteinen, die den Kontakt des Computers mit der übrigen Welt erlauben, also auch mit uns. Dieses Aufbauprinzip finden wir bei allen 8-Bit-Computern.

5. Das Innenleben eines Mikroprozessors

Um es gleich noch mal zu sagen: Was hier erzählt wird, ist nicht dazu geeignet, Elektronik-Freaks den totalen Durchblick zu geben. Wenn Sie das aber gerne möchten, dann sehen Sie sich zum Beispiel die Blockschaltbilder an im »Programmer's Reference Guide« für den Commodore 64 auf Seite 404 oder im »MOS-Hardware-Handbuch« auf Seite 34. Auch Rodney Zaks' Buch »Programmierung des 6502« ist zu empfehlen. Er hat sich viel Mühe gegeben, sich verständlich auszudrücken. Mir kommt es nur auf den allgemeinen Überblick an. Den sollen Sie bekommen, wenn wir uns jetzt zusammen Bild 2 betrachten.

Da sehen Sie zunächst als Herzstück des Prozessors, die ALU (Arithmetik Logical Unit), also den arithmetisch-logischen Baustein. Die ALU hat die Fähigkeit, Rechenoperationen auszuführen mit Daten, die sie über den Datenbus und normalerweise vom Akkumulator erhält. Das Ergebnis wird ebenfalls im Akkumulator abgelegt (daher auch der Name: von akkumulieren, etwas ansammeln). Der Akkumulator ist das Register, das uns als Programmierer am häufigsten beschäftigen wird. Er ist die Sammel- aber auch die Verteilerstelle für fast alle Daten, die wir hin-

Statusflaggen-Register (hier P genannt). Man findet darin angezeigt, ob eine Rechenoperation ein negatives Ergebnis hatte, ob eine Null aufgetaucht ist oder ob ein Übertrag stattgefunden hat. Auch dieses Register wird uns noch häufig begegnen. Das Stapelregister, auch Stackpointer (Stapelzeiger) genannt, gibt uns Auskunft über den Füllungsgrad eines 256 Byte großen speziellen Speichers, der vom Prozessor direkt verwaltet wird. Auch damit werden wir noch oft zu tun haben. Schließlich kommen wir zur vorhin erwähnten Ausnahme, zum Programmzähler (PCL, PCH). Das ist ein 16-Bit-Register, das sich aus zwei 8-Bit-Registern (PCL für das LSB und PCH für das MSB) zusammensetzt und daher alle 65535 Speicherplätze ansprechen kann. Hier ist immer die Adresse des nächsten abzuarbeitenden Befehls enthalten.

Ich will an dieser Stelle nicht in die Einzelheiten der Befehlsabarbeitung einsteigen (das können Sie auch bei Rodney Zaks nachlesen, wenn Sie es genau wissen wollen). Es soll nur gesagt sein, daß sich die Verarbeitung in drei Schritte unterteilen läßt:

a) den nächsten Befehl holen
b) den Befehl decodieren
c) den Befehl ausführen

Zu c) ist noch zu sagen, daß es Befehle gibt, die der Prozessor ohne weitere Angaben ausführen kann. Für andere müssen erst noch weitere Daten aus dem Speicher geholt oder dort abgelegt werden. Deswegen brauchen die Befehle unterschiedliche Zeiten zur Ausführung. Die Zeit wird als Anzahl von sogenannten Taktzyklen in den Befehlstabellen angegeben. Unser Computer hat eine Taktfrequenz von rund 1 MHz, was bedeutet, daß ein Taktzyklus etwa eine Mikrosekunde (10^{-6} Sekunden) dauert.

Auf diese Weise wurde die Zeitdauer für unser kleines Demonstrationsprogramm zu Anfang berechnet. Auch das werden Sie noch lernen.

6. Der Speicher unseres Computers: eine Straße mit 65536 Hausnummern

Ein weiteres »lebenswichtiges Organ« unseres Computers ist der Speicher, mit dessen Aufbau wir uns jetzt auseinandersetzen werden.

Stellen Sie sich eine lange Straße vor mit 65536 aneinandergereihten Häusern (von Hausnummer 0 bis Hausnummer 65535, wie in Bild 3)

Dies entspricht unserem Speicher. Jedes Haus (Byte) ist ebenerdig und hat acht Zimmer (Bits). Wie eine Stadt in Stadtteile unterteilt ist, finden wir in dieser Speicherstadt die Einteilung in Pages english »Seite«. Jede Page besteht aus 256 aufeinanderfolgenden Häusern. Ähnlich, wie es in Städten ein Handwerkerviertel, ein Geschäftsviertel und so weiter gibt, sind auch hier manchen Pages spezielle Aufgaben zugeteilt. Die wichtigste davon ist die Zeropage (also die Page 0), auf der sich die CPU beziehungsweise das Betriebssystem Notizen macht. Auch die Pages 1 bis 3 (also bis zur Hausnummer 1023) dienen ähnlichen Zwecken. Ab Page 4 bis inklusive Page 7 liegt der Bildschirmspeicher unseres Computers. Er entspricht genau dem, was auf dem Bildschirm zu sehen ist. Jedes Zeichen ist dabei durch einen POKE-Code vertreten. Die Zuordnung der einzelnen Adressen zu den Bildschirmpositionen kann man aus dem Handbuch entnehmen, ebenso auch die POKE-Codes.

Wenn wir also in die Adresse 1024 eine 1 durch:

```
POKE 1024,1
```

schreiben, dann erscheint in der oberen linken Bildschirmecke der zum POKE-Code 1 gehörige Buchstabe, nämlich das A. Bei Ihnen zeigt sich kein A? Dann fahren Sie mal mit dem Cursor an die Stelle und Sie erkennen den Buchstaben. Den C64 gibt es mittlerweile in verschiedenen Versionen des Betriebssystems. Bei der älteren muß man außer dem Bildschirmcode (POKE-Code) in den Bildschirmspeicher auch noch einen Farbcode in die entsprechende Bildschirm-Farbspeicherstelle geben. Diese kann man ebenfalls dem Handbuch entnehmen. Besitzer älterer C64-Modelle müssen also noch den Befehl

```
POKE 55296,1
```

hinzufügen, um das A in weißer Farbe sichtbar zu machen. Die neueren Versionen verlangen diese Farbbefehle nicht mehr, oder nur noch dann, wenn man eine andere Farbe als die vorgegebene erhalten möchte.

Der Bildschirmspeicher erfordert genau

```
25 mal 40 = 1000 Byte.
```

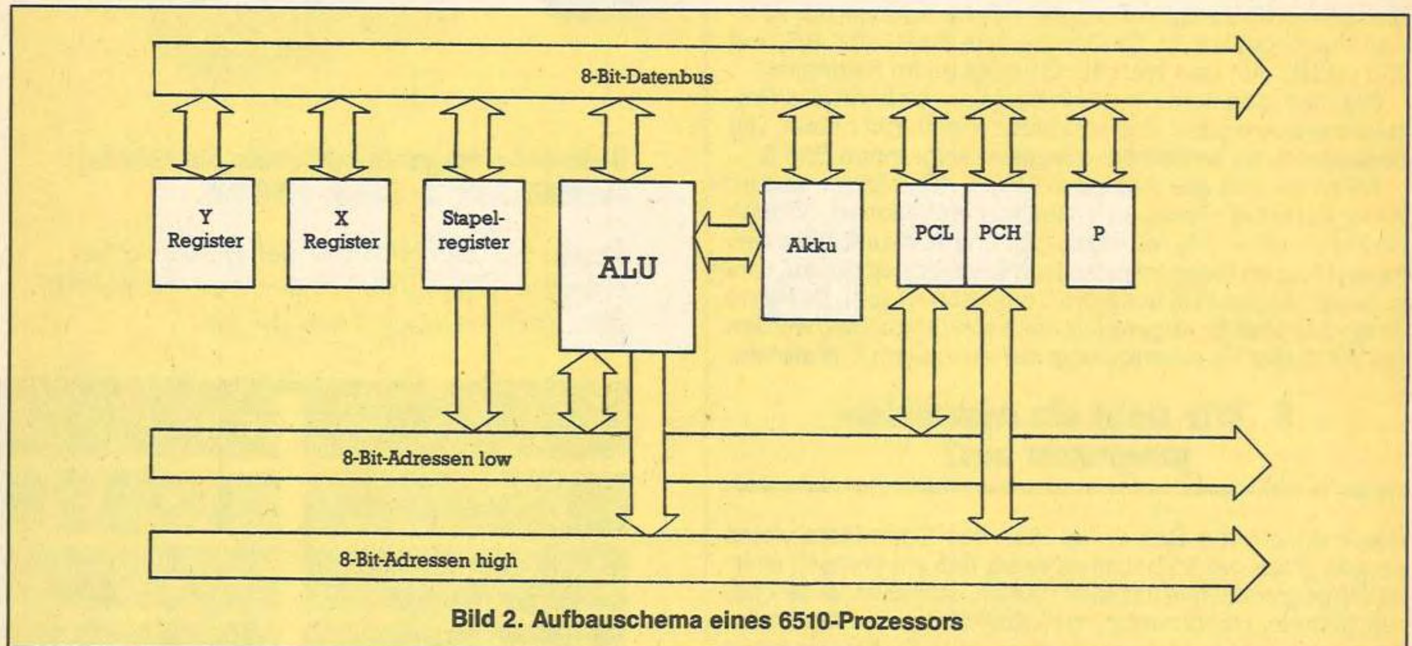


Bild 2. Aufbauschema eines 6510-Prozessors

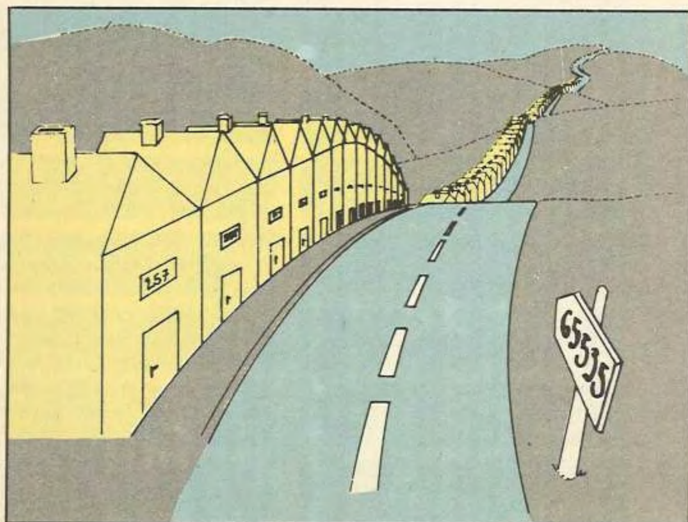


Bild 3. Die Speicherstadt im C 64

Von den 1024 Byte (4 Pages) sind also noch 24 Byte frei, die teilweise als Sprite-Zeiger Verwendung finden. Doch dazu kommen wir später. Von der Page 8 an, also der Hausnummer 2048, haben wir volle Verfügungsgewalt über den Speicher für Basic-Programme und Daten.

Etwas Neues passiert ab Hausnummer 40960, dem Ende unseres Basic-Speichers: Von dieser Adresse an, bis 49151, haben die Gebäude der Speicherstadt eine zusätzliche erste Etage. Zu ebener Erde liegt weiterhin RAM vor, im ersten Stock aber ROM, und zwar der 8 KByte große Basic-Interpreter (Bild 4 auf Seite 16).

In den nächsten 4 KByte finden wir wieder ausschließlich RAM. Dieser Bereich (von 49152 bis 53247) wird häufig für Maschinenprogramme genutzt, weil hier nicht die Gefahr des unabsichtlichen Überschreibens durch Basic-Programme oder Variable besteht. Ab Adresse 53248 sind die Byte-Häuser sogar mit zwei Etagen versehen. Im Erdgeschoß liegt weiterhin RAM, in der ersten Etage sind die Ein- und Ausgabebausteine angesiedelt, und oben im zweiten Stockwerk breitet sich das Zeichen-ROM aus. Die Bele-

gung im ersten Geschoß durch die Ein- und Ausgabebausteine ist in Bild 4a zu erkennen.

Von der Adresse 57344 an bis zum Ende unserer Speicherstadt haben wir es wieder mit eingeschossigen »Byte-Häusern« zu tun, in deren Erdgeschoß sich RAM und in deren erster Etage sich das Betriebssystem-ROM aufhält. Wie man die verschiedenen Etagen in das Blickfeld eines Programms – insbesonders eines Assembler-Programms – rücken kann, werden wir im Kapitel 35 erfahren.

7. Auskunft über das Befinden unseres Computers: die Register-Anzeige

Bisher haben wir uns mit dem Innenleben unserer Computer auseinandergesetzt und die wichtigsten Teile der Hardware kennengelernt. Jetzt kommen wir zur Software, nämlich zum Assembler. Wenn Sie jetzt den SMON (siehe Seite 132) einschalten, meldet er sich mit einer Registeranzeige (Bild 5 auf Seite 16).

Die angezeigten Werte sind Beispiele, wie sie beim C 64 auftreten können. PC ist der Programmzähler, der immer auf den nächsten zu holenden Befehl zeigt. (Der Wert $E147 rührt vom SYS-Aufruf, mit dem ich meinen Assembler starte.) IRQ zeigt uns an, auf welche Adresse der Interrupt-Vektor gestellt ist. Das ist das Byte-Paar 788 (LSB) und 789 (MSB). Auf den Wert $EA31 zeigt es im Normalfall.

Die nächsten acht Angaben beziehen sich auf das Prozessorstatusregister, das wir zuletzt P genannt haben. Die Bedeutung der einzelnen »Flaggen« zeigt Ihnen Bild 6.

AC ist der aktuelle Inhalt des Akkus. XR zeigt an, was im X-Register und YR was im Y-Register enthalten ist. SP (von Stackpointer = Stapelzeiger) gibt uns Auskunft über den freien Platz im Stapelregister. Damit wissen wir genau, was in diesem Moment in unserem Computer vorgeht. So fremd Ihnen das alles im Augenblick noch vorkommt, bald werden Sie mit dieser Registeranzeige auf vertrautem Fuß stehen.

8. Wie sieht ein Assemblerprogramm aus?

Das menschliche Gehirn hat dem des Computers vieles voraus. Dazu gehört beispielsweise, daß ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, Musik hören, lächeln, Handbewegungen ausführen, womöglich dabei auch noch etwas kauen und so weiter. Ein Computer ist dazu nicht imstande. Er erledigt eine kleine Aufgabe nach der anderen. Weil er das so schnell macht, hat es für uns den Anschein, es geschähe alles gleichzeitig. Das Maschinenprogramm ist eine Kette solcher kleiner Aufgaben. Das erste Glied daraus, das wir kennenlernen wollen, ist der Befehl

LDA.

Das bedeutet: Lade den Akkumulator. Alle Assembler-Befehlsworte bestehen aus drei Buchstaben wie dieser hier auch. Wir haben schon erwähnt, daß einem solchen Befehl eine 8-Bit-Codezahl entspricht. Das ist hier $A9 oder binär 1010 1001 oder schließlich dezimal 169. Die Codezahl muß in einem Speicherplatz stehen, zum Beispiel in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann so aus:

1500 LDA

Hier tritt also die Speicherplatznummer mit einem nachfolgenden Befehl anstelle der von Basic gewohnten Zeilennummer.

Es fehlt noch etwas Entscheidendes: Was soll denn in den Akku geladen werden? Genauso wie es in Basic Befehle gibt, die für sich alleine stehen können wie CLR oder

64'er Sonderhefte im Überblick

Die 64er-Sonderhefte bieten Ihnen detaillierte Informationen zu speziellen Themen rund um die Commodore-Computer.

Bestellen Sie bitte die gewünschten Ausgaben zum Preis von jeweils 14,– DM mit der Zahlkarte auf Seite 34.

C 64-Einstieg



SONDERHEFT 0005: C 64-GRUNDWISSEN
Vom ersten Einschalten bis zum eigenen Programm / Grundlagen, Tips und Tricks



SONDERHEFT 0016: EINSTEIGER 2
Spriteanimation: Zeichentrickfilm mit dem Computer / GEOS, die neue Benutzeroberfläche



SONDERHEFT 0019: EINSTEIGER 3
Basic-Kurs / Programm-Übersicht

C 128



SONDERHEFT 0001: C 128
Das können C 128 und C 128 D / Vergleich: C 128–C 64 / die passende Peripherie



SONDERHEFT 0010: C 128 II
Die Geheimnisse von CP/M / Kompletter C 128-Schaltplan / Grafik für Einsteiger



SONDERHEFT 0022: C 128 III
Farbiges Scrolling im 80-Zeichen-Modus / 8-Sekunden-Kopierprogramm

Tips & Tricks, Anwendungen



SONDERHEFT 9901: TIPS & TRICKS
Befehlserweiterungen für Betriebssystem und Floppy / Unentbehrliche Programmierhilfen



SONDERHEFT 0002: TIPS & TRICKS
Zeichensatz- und Sprite-Editor / Interrupt-Joystickabfrage / 27 nützliche Einzeiler



SONDERHEFT 0024: TIPS, TRICKS & TOOLS
Automatische Textkorrektur / Utilities / Basic-Compiler zum Abtippen



SONDERHEFT 9907: ANWENDUNGEN/DFÜ
Terminal- und Mailboxprogramm zum Abtippen / Der C 64 als Winzer

Spiele



SONDERHEFT 9902: ABENTEUERSPIELE
45 Seiten Adventure-Programmier-Kurs / Listings und Schritt-für-Schritt-Lösungen



SONDERHEFT 0004: ABENTEUERSPIELE
Kurs: Programmierung von Grafik, Parser und Künstlicher Intelligenz / Viele Adventures



SONDERHEFT 9903: SPIELE
Top-Spiele-Listings für C 64 und VC 20 / Große Spiele-Marktübersicht



SONDERHEFT 0017 SPIELE FÜR C 64 UND C 128
So programmiert man Scrolling / Strategiespiele: Grips ist gefragt

Drucker, Grafik, Sound



SONDERHEFT 0018: DRUCKER
Listing: professionelle Textverarbeitung für den MPS 801 / Matrixdrucker im Test



SONDERHEFT 9904: GRAFIK & DRUCKER
80-Zeichen-Karte zum Abtippen / Hardcopy-Routinen für viele Drucker



SONDERHEFT 0006: GRAFIK
Giga-CAD: 3-D-Konstruktionsprogramm / Grafikprogrammierung von C 64 und C 128



SONDERHEFT 0023: GRAFIK / ANWENDUNGEN
Paint Magic: ein professionelles Malprogramm



SONDERHEFT 0020: GRAFIK
Grafik-Programmierung / Bewegungen

C16, C116, Plus/4



SONDERHEFT 0003: C 16/116, VC 20
Grundlagen: Grafik und Soundprogrammierung mit dem C 16 / Listings: Anwendungen, Spiele.



SONDERHEFT 0008 PLUS/4 UND C 16
Übersicht: Zeropage und wichtige Systemadressen / Grundlagen und viele Listings

Programmiersprachen, Maschinensprache



SONDERHEFT 0012: PROGRAMMIER-SPRACHEN
Pascal, Comal, Prolog, C und Forth / Vergleich: Basic-Compiler



SONDERHEFT 0021: ASSEMBLER UND BASIC
Giga-Ass: Hypra-Ass hoch 2 / Paradoxon-Basic: 50000 Basic Bytes free



SONDERHEFT 0007: PEEKs UND POKEs
Die wichtigsten Speicherstellen von C 64, C 128 und C 16 / Listings: Tips & Tricks

Floppy, Datasette, Dateiverwaltung



SONDERHEFT 0025: FLOPPY / DATASETTE
Kurse: Floppy-Programmierung für Einsteiger und Profis



SONDERHEFT 0009: FLOPPY & DATEI-VERWALTUNG
Floppy-Beschleuniger im Vergleichstest / Arbeiten mit dBase II / C 128-Diskmonitor



SONDERHEFT 0015: FLOPPY & DATASETTE
Reparaturanleitung: Erste Hilfe für die Diskettenstation / Hypratape: das Super-Turbotape

Hardware



SONDERHEFT 0013: HARDWARE
Ein-Chip-Mikrocomputer / Bauanleitungen: MIDI-Interface, Speicheroszilloskop, IC-Tester

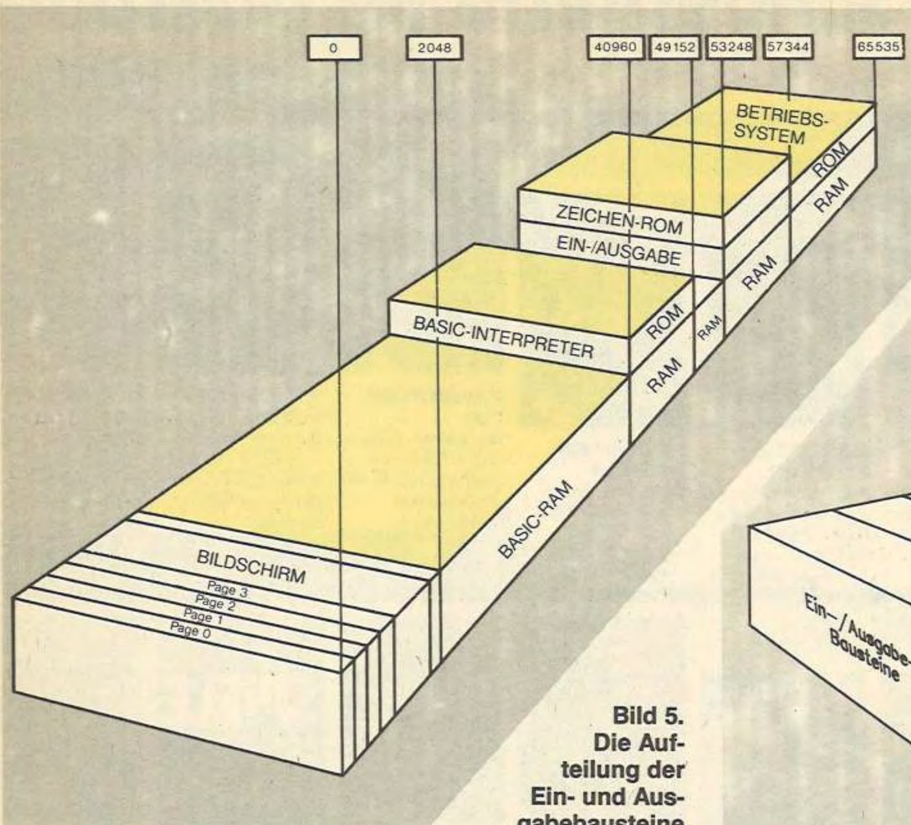
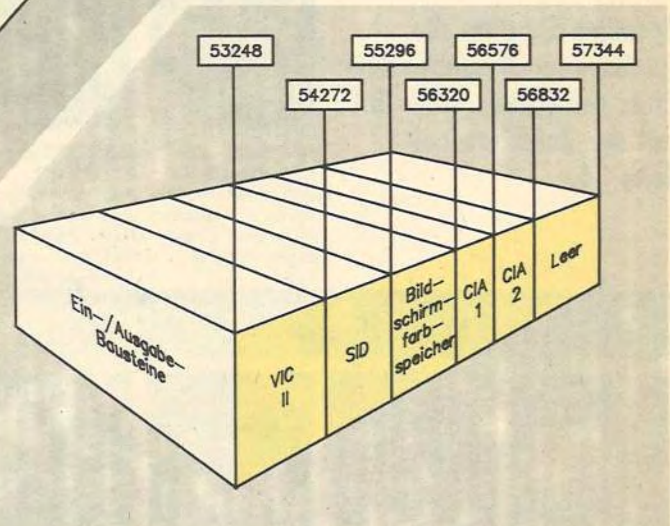


Bild 5. Die Aufteilung der Ein- und Ausgabebausteine

Bild 4. Die Speicherarchitektur des C 64 auf einen Blick



LIST, gibt es auch im Assembler solche Befehle. Weitaus häufiger sind aber hier Befehle, die ein »Argument« erfordern (in Basic zum Beispiel PEEK(100). Dabei ist 100 das Argument). In Assembler gibt es zwei Sorten von Argumenten. Solche, die in einem Speicherplatz unterzubringen sind und andere, die zwei Byte brauchen. Mit dem Befehlswort (hier also LDA) zusammengezählt, existieren in Assembler also 1-Byte-Befehle, 2-Byte-Befehle und 3-Byte-Befehle.

Das Argument von LDA ist also das, was in den Akku soll. Laden wir deshalb mal eine 1 in den Akku:

1500 LDA #$01

Wir haben jetzt einen 2-Byte-Befehl erzeugt. Was aber bedeuten »#« und »$« dabei? $ ist leicht zu erklären. Die große Mehrzahl der Assembler nimmt bei Zahlenangaben Hexadezimalzahlen an. Bei einigen muß man dies durch das $-Zeichen kennzeichnen. Manche Assembler lassen auch Binärzahlen, Dezimalzahlen und sogar ASCII-Zeichen als Argumente zu. Für jede Eingabeart steht dann vor dem Argument ein Zeichen, das die Art des Argumentes angibt, zum Beispiel häufig »!« für Dezimalzahlen oder »%« für Binärzahlen. Nun zum #-Zeichen. Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl – wie wir hier —, aber zum Beispiel auch mit dem Inhalt eines anderen Speichers und so weiter. Man spricht dabei von der sogenannten Adressierung.

Es gibt eine ganze Menge davon und jede wird auf eindeutige Weise gekennzeichnet. Wenn wir in unserem Akku eine Zahl laden, dann ist das die »unmittelbare« Adressierung und die kennzeichnet man mit dem #-Zeichen.

Wenn in Speicherstelle $1500 die Codezahl für LDA steht, dann muß die 1 in der Speicherstelle $1501 stehen, wie es sich für einen 2-Byte-Befehl gehört. Wenn Sie nun die Assemblerzeile eingegeben haben und ⟨RETURN⟩ drücken, dann taucht auf dem Bildschirm eine Fehlermeldung auf (bei vielen Assemblern). Wir müssen vorher nämlich noch unserem Software-Instrument sagen, jetzt zu assemblieren. Wie das geschieht, ist auch wieder von Assembler zu Assembler verschieden. Die meisten erwarten, daß man vor der Zeile noch ein A eingibt (zum Beispiel beim C 128):

A 1500 LDA #$01

Wenn Sie jetzt ⟨RETURN⟩ drücken, zeigt der Bildschirm:

A 1500 LDA #$01
A 1502

und meistens einen blinkenden Cursor, der auf die nächste Eingabe wartet. $ 1502 ist die nächste freie Speicherstelle, und wenn beim Programmablauf der Programmzähler nach dem LDA #$01 auf $1502 deutet, dann erwartet er dort den nächsten Befehl. Wenn dort Unsinn steht, dann stürzt der Computer im allgemeinen ab, je nachdem, welcher Code dann hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten dafür: $00 bis $FF. Im Gegensatz zu Basic, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen wie man will, muß hier das Programm eine ununterbrochene Perlenschnur von Befehlen in Speicherstellen sein. Durch einige Befehle läßt sich dieses Prinzip allerdings durchbrechen.

Damit wir die Wirkung von Befehlen sehen können, greife ich auf einen Befehl vor, der ähnlich dem STOP in Basic einen Programmabbruch bewirkt: BRK. Die genaue Funktion soll erst später erklärt werden, aber wir sehen jedenfalls dann, wenn ein Maschinenprogramm auf einen BRK-Befehl läuft, die Registerinhalte angezeigt. Das ist in den meisten Assemblern eingebaut. Wir ergänzen jetzt:

A 1502 BRK

Damit erstmal genug. Steigen Sie aus dem Assembler aus und starten Sie das Programm. In den meisten Assemblern geht das mit

G 1500

oder sonst von Basic aus mit SYS 5376. Jetzt werden wieder die Register angezeigt. Der Programmzähler steht auf 1503, im Akku steht 01, alle Flaggen außer der Breakflagge sind Null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument:

A 1500 LDA #$00
A 1502 BRK

Wir starten wieder und sehen uns die Register an: Pro-

grammzähler 1503, Akku jetzt 00, aber bei den Flaggen hat sich etwas verändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen also: Diese Flagge bleibt solange ungesetzt, so lange nicht eine Null im Akku auftaucht, erst dann wird sie 1.
Noch einmal ändern wir das Programm:

```
A 1500 LDA #$FF
A 1502 BRK
```

Nach erneutem Start steht das Erwartete in den Registern, nur bei den Flaggen ist etwas Merkwürdiges passiert: die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Nun wissen wir aber, daß $FF = dez. 255 ist. Dieses Rätsel wird uns noch eine Weile begleiten. Es sei hier nur bemerkt, daß kein Fehler vorliegt: Immer wenn in einer Zahl das Bit 7 gleich 1 ist, geht die Vorzeichenflagge auf 1. Die Lösung des Rätsels werden wir bei den negativen Binärzahlen finden.

Wir schließen aus alledem: Der LDA-Befehl beeinflußt die Vorzeichen- und die Zeroflagge.

9. Die absolute Adressierung

STA heißt »STore Accumulator«, also »lege Akkuinhalt ab«. Wie Sie sich denken können, muß auch hier ein Argument auftauchen: nämlich wohin abgelegt werden soll. Wir legen unseren Akkuinhalt in die erste Bildschirmspeicherstelle (C 64:$0400). Unser Programm muß also so aussehen:

```
A 1500 LDA #$01
A 1502 STA $0400
```

PC	IRQ	NV-BDIZC	AC	XR	YR	SP
E147	EA31	10110000	00	00	00	F8

Bild 5. Eine Registeranzeige

N	V	—	B	D	J	Z	C
Negativ-Flagge	Über-lauf-Flagge	unbe-nutzt	Abbruch-Flagge	Dezimal-Flagge	Interrupt-Flagge	Zero-(Null) Flagge	Carry-(Über-trag) Flagge

Bild 6. Das Prozessor-Status-Register P: die Flaggen

Mit diesem STA-Befehl lernen wir eine neue Adressierungsart kennen: die »absolute« Adressierung. Sie ist daran zu erkennen, daß kein besonderes Merkmal verwendet wird. Die Adresse $ 0400 ist nicht in einem Byte darstellbar, sondern wird aufgeteilt auf zwei Bytes. Im Speicher steht jetzt:

```
1500 LDA #
1501 $ 01
1502 STA
1503 $ 00        »das ist das LSB«
1504 $ 04        »das ist das MSB«
```

Hier liegt also ein 3-Byte-Befehl vor und die nächste freie Speicherstelle ist $ 1505.

Vom Basic her wissen Sie, daß 1 der Bildschirmcode für den Buchstaben A ist und daß man jeder Bildschirmspeicherstelle auch eine Bildschirmfarbspeicherstelle zuordnet. Um ein eingeschriebenes Zeichen vom Hintergrund abzuheben, muß man dort dann bei den älteren C 64-Versionen eine Farbinformation eingeben. Der Start dieses Bildschirmspeichers liegt beim C 64 an folgender Adresse:

$ D800

Der Farbe Schwarz entspricht die Codezahl 0. Wir ergänzen unser Programm durch:

```
A 1505 LDA #$00
A 1507 STA $D800
```

Die nächste freie Adresse ist nun $150A. Unser Programm soll jetzt abgeschlossen sein. Damit der Computer aber beim Programmzählerstand $150A nicht Unsinn vorfindet, muß - ähnlich wie bei END in Basic - das Programm auf irgendeine Weise beendet werden. Das kann durch BRK geschehen. Wir wollen aber den dritten Assembler-Befehl kennenlernen:

RTS

Das heißt »Return From Subroutine«, also »Rückkehr aus Unterprogramm«. In unserem Fall bewirkt das eine Rückkehr zum Basic. Wie Sie sehen, ist das ein 1-Byte-Befehl, also ohne Argument. Auch hier spricht man von einer Adressierungsart, nämlich der »impliziten«-Adressierung. Man erkennt sie am Fehlen des Argumentes. Die Adresse ist implizit, das heißt im Befehl selbst enthalten. Dies ist nämlich ein Befehl, der immer an den Programmzähler gerichtet ist. Der Computer holt sich vom Stapel-Speicher die dort zuoberst liegende Adresse, das ist die, bei der der Computer in ein Unterprogramm gesprungen ist oder aber die, bei der der Computer Basic verlassen hat. Wir ergänzen also noch:

```
A 150A RTS
```

und starten das Programm, zum Beispiel von Basic aus mit SYS 5376. Natürlich taucht dann in der linken oberen Ecke des Bildschirmes ein schwarzes A auf. Hier noch der Basic-Lader:

```
10 FOR I=5376 TO 5386:READ A:POKE I,A:NEXT I:END
20 DATA 169,1,141,0,4,169,0,141,0,216,96.
```

10. Vier neue Befehle

Eine Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl in Basic. Man kann in Assembler nicht direkt eine Zahl in einen Speicher einschreiben, sondern muß den Umweg über den Akku machen. Außer dem Akku eignen sich dazu aber auch das X-Register und das Y-Register. Hierfür gibt es die Befehle LDX (lade X-Register), STX (lege X-Register-Inhalt ab), LDY (lade Y-Register) und schließlich STY (lege Y-Register-Inhalt ab). Sie können das übungshalber an unserem kleinen Programm ausprobieren. An dem folgenden Programm sehen Sie noch eine Eigenart der drei Register (Akku, X-Register, Y-Register):

```
A 1500    LDA    #$01
A 1502    LDX    #$00
A 1504    LDY    #$02
A 1506    STA    $0400
A 1509    STX    $D800
A 150C    STY    $0401
A 150F    STX    $D801
A 1512    STA    $0402
A 1515    STX    $D802
A 1518    RTS
```

Dieses Programm druckt - wie erwartet - »ABA« in die linke obere Ecke des Bildschirms. Dabei ist das X-Register dreimal ausgelesen worden und der Akku zweimal. Sie sehen also, daß die Registerinhalte durch die STA-, STX-, STY-Befehle nicht verändert werden.

Wir wollen noch etwas ausprobieren. Bisher haben wir den LDA-Befehl nur mit der »unmittelbaren« Adressierung kennengelernt. LDA, LDX, LDY können auch »absolut« adressiert werden.

```
A 1518 LDA $D800
```

Damit laden wir den Inhalt der Speicherstelle $ D800 in den Akku. Der Inhalt ist seit $1509 eine Null. Jetzt weiter:

```
A 151B STA $0403
A 151E STX $D803
A 1521 RTS
```

Das müßte beim Ablauf des Programms noch einen Klammeraffen (@mit Bildschirmcode 0) an die vierte Stelle plazieren, was Sie durch SYS 5376 leicht nachprüfen können. Sie sehen, daß man mit diesen sieben Befehlen schon eine Menge anfangen kann.

Wir kommen noch einmal zur Adressierung. Ich hatte Ihnen gesagt, daß LDA #$01 ein 2-Byte-Befehl mit unmittelbarer Adressierung ist (ein Byte für LDA und eines für 01), LDA $D800 ist ein 3-Byte-Befehl (ein Byte für LDA, je eines für das LSB und das MSB von $D800) mit absoluter Adressierung. Da werden Sie sich schon gefragt haben, wo bleibt die Adressierung! Wenn aber kein Byte für die Adressenmarkierung (zum Beispiel #) reserviert ist, muß die Kennzeichnung irgendwie anders sein. Wenn Sie einen Disassembler zur Verfügung haben, dann sehen Sie sich damit unser Programm an. Fast jeder Disassembler gibt neben dem Assemblertext auch Byte für Byte in Hexadezimalzahlen die Codes an. Wenn Sie nun die beiden Befehle LDA #$01 und LDA $d800 von den Codes her untersuchen, sehen Sie folgendes:

```
1500 A9 01       LDA#$01
```

und

```
1518 AD 00 D8   LDA $D800
```

Offensichtlich gehört jeweils das erste angezeigte Byte zu LDA. Sie sind aber verschieden! Wir sehen daraus, daß die Codezahl für einen Befehl gleich zwei Informationen enthält: das Befehlswort selbst (LDA) und die Adressierungsart.

Genauso wie man LDA sowohl unmittelbar als auch absolut ausführen kann, ist das auch mit LDX und LDY möglich. Bei den Befehlen STA, STX, STY ist eine unmittelbare Adressierung sinnlos. Für RTS kennt man nur eine implizite Adressierung. Wir fassen das alles in Bild 7 zusammen.

In den letzten Spalten von Bild 7 ist noch angegeben, inwieweit durch diese Befehle das Prozessorstatusregister beeinflußt wird, so wie wir es für den Befehl LDA schon ausprobiert haben. In der vorletzten Spalte sehen Sie, wie lange die Ausführung eines Befehls dauert. Wenn ein Taktzyklus etwa eine Mikrosekunde dauert, dann müßten Sie jetzt ausrechnen können, wie lange unser letztes Programm zur Bearbeitung braucht: 48 Mikrosekunden. Ein vergleichbares Basic-Programm braucht dazu etwa tausendmal so lange: zirka 0,05 Sekunden.

Befehlswort	Adressierung	Byteanzahl	Code HEX	Code DEZ	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	unmittelbar	2	A9	169	2	N, Z
	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Bild 7. Die ersten sieben Befehle

11. Die Zahlen der Assembler-Alchimisten

Ein bißchen von Assembler-Alchimie verstehen Sie jetzt schon mit diesen sieben Befehlen. Wir wollen uns nun die Zahlen ansehen, die hier Verwendung finden: das Binärsystem und das Hexadezimalsystem.

Die einzigen Ziffern, die unser Computer kennt, sind 0 und 1. Sie stehen für »Strom aus« oder »Strom an«, oder für »keine magnetische Erregung« oder »magnetische Erregung«. Deshalb ist es für uns als angehende Assembler-Alchimisten von großer Bedeutung - wir arbeiten ja ganz eng an der Hardware - dieses binäre Zahlensystem handhaben zu können. Das Hexadezimalsystem kennt der Computer eigentlich gar nicht. Wir verwenden es deswegen, weil es in einem besonders engen Zusammenhang mit Binärzahlen und dem Aufbau unseres Computers steht: Die größte einstellige Hex-Zahl ist $F, das entspricht genau 1111 im Binärsystem, also dem maximalen Füllungsgrad eines halben Bytes, das Nibble genannt wird. Ein ganzes Byte kann maximal $FF enthalten (binär 1111 1111) und der gesamte Speicheradressenbereich unseres Computers geht bis $FFFF (dezimal 65535). Eine einstellige Hex-Zahl paßt also in ein Nibble, eine zweistellige in ein Byte und eine dreistellige oder vierstellige in zwei Byte, weshalb man solche Hex-Adressen auch recht leicht in das LSB und das MSB (auch Low- und High-Byte genannt) aufteilen kann:

```
$  D8  00
   MSB LSB
```

Rechnen werden wir mit Hexadezimalzahlen nicht, dazu benutzen wir dann das Dezimalsystem oder - wenn es sich um computerinterne Vorgänge handelt - das Binärsystem.

Das Rechnen mit Binärzahlen funktioniert genauso wie das mit Dezimalzahlen. Es gilt also

```
0+0=0
0+1=1
1+0=1
1+1=10
```

wobei binär 10 gleich dezimal 2 ist. Als Beispiel können wir mal 2+1=3 im Binärsystem rechnen:

```
  10 entspricht dez. 2
+ 01 entspricht dez. 1
----------------------
  11, was ja dezimal 3 ergibt.
```

Die Addition erfolgt also spaltenweise wie beim gewohnten dezimalen Addieren. Auch mit dem Übertrag läuft es wie im dezimalen. Beispiel: 2+2=4:

```
  10 entspricht dez. 2
+ 10 entspricht dez. 2
----------------------
 100, was dezimal eine 4 ergibt.
```

In der zweiten Spalte wurde nach der Regel verfahren: 1+1=10. Rechnen wir noch 3+3=6:

```
  11 entspricht dez. 3
+ 11 entspricht dez. 3
----------------------
 110, was dezimal eine 6 ergibt.
```

In der ersten Spalte wurde gerechnet 1+1=10, wobei nach dem alten Motto: »0 hin, 1 im Sinn« die 0 unter den Strich gesetzt wurde. In der zweiten Spalte wird dann so verfahren: 1+1+1 (das ist die 1, die wir »im Sinn« hatten)=11. Ich meine, daß Sie ohne Probleme die folgenden Übungsaufgaben lösen und dann jeweils dezimal das Ergebnis nachprüfen können: 10+5, 7+1, 16+16, 240+16, 62+65.

12. Eine Zauberformel der Assembler-Alchimisten: INX, INY, INC, DEX, DEY, DEC?

Wir wissen ja schon, daß man diese »Zauberformeln« entzaubern kann. INX heißt einfach »INcrement X-Register«, also Inhalt des X-Registers um 1 erhöhen. Es wird Ihnen sicher einleuchten, daß INY dasselbe mit dem Y-Register tut. Etwas weniger deutlich ist das bei INC. Das bedeutet »INCrement memory«, also zähle zum Inhalt einer Speicherstelle eins dazu. INX und INY enthalten alles, was dem Computer zu sagen ist, sind also offensichtlich 1-Byte-Befehle mit der in der letzten Folge schon kennengelernten impliziten Adressierung. Bei INC muß dem Computer noch gesagt werden, welche Speicherstelle er um 1 erhöhen soll. Es gehört also noch eine Adresse dazu. Das läßt diesen Befehl im allgemeinen zu einem 3-Byte-Befehl werden.

Das Umgekehrte leisten die Befehle DEX, DEY und DEC. Sie bedeuten nämlich »DECrement X-Register«, also »zähle das X-Register um eins herunter«, beziehungsweise das Y-Register oder - bei DEC - die angegebene Speicherstelle. Für die Adressierungsart und die Anzahl Bytes pro Befehl gilt hier das gleiche wie für die INX...-Befehle. Sehen wir uns das an einem kleinen Beispiel an:

```
1500  LDA #00
1502  LDX #01
1504  STA $ D800
1507  STX $ 0400
150A  INX
150B  STA $ D801
150E  STX $ 0401
1511  DEX
1512  STA $ D802
1515  STX $ 0402
1518  BRK
```

Wenn Sie das kleine Programm mit G 1500 starten, dann sollten Sie in der linken oberen Ecke des Bildschirms ABA in schwarzer Schrift stehen haben. Was ist geschehen? Wir haben den Inhalt des Akku (=0, also Farbcode für schwarz) in das Bildschirm-Farbregister geschrieben (#D800), dann den Inhalt des X-Registers (1 = POKE-Code für den Buchstaben A) in die erste Bildschirm-Speicherzelle (#0400). Anschließend wurde das X-Register um 1 erhöht (2 = POKE-Code für den Buchstaben B) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem mußte natürlich auch dieser Bildschirm-Farbspeicherplatz mit dem Farbcode 0 belegt werden. Durch DEX wurde das X-Register wieder herunter gezählt, somit wieder ein A erzeugt und in die dritte Bildschirmstelle gedruckt.

Sie haben sicher schon bemerkt, daß man auf diese Weise Abläufe mitzählen kann. Soll zum Beispiel ein Vorgang 20mal wiederholt werden, dann schreibt man ins X-Register (oder ins Y-Register oder in eine andere Speicherstelle) den Anfangswert 0, läßt den Computer eine Arbeit ausführen, erhöht das entsprechende Register oder die Speicherzelle um 1 mit INX, INY oder INC, prüft dann, ob dieser Inhalt schon 20 geworden ist und so weiter. Wie man diese Prüfung vornimmt, dazu kommen wir erst später bei den BRANCH-Befehlen. Das ist also ähnlich wie in Basic bei den FOR...NEXT-Schleifen: Dort wird eine Variable als Zähler verwendet, hier ein Register (oder eine Speicherstelle). Ebenso wie in Basic bei diesen Schleifen kann man auch hier rückwärts zählen mit DEX, DEY oder DEC. Das hat oft gewisse Vorzüge, was uns aber noch nicht kümmern soll.

Wenn wir diese Befehle als Zähler verwenden, sollten wir im Auge behalten, daß eine Speicherstelle (auch ein X- oder Y-Register) Zahlen nur von 0 bis 255 enthalten kann. Die höchste 8-Bit-Zahl ist ja:

```
dez. 255 =  bin. 1111 1111
      +1                 1
            --------------
ergibt:     (1) 0000 0000
```

Wenn wir also über 255 hinauszählen, ergibt sich wieder 0 und so weiter, weil ein Überlauf stattgefunden hat. Das 9.Bit paßt nicht mehr in das Byte hinein. Um noch mal ge-

nau sehen zu können, was unser Computer da tut, probieren Sie einmal aus:

```
1500 LDA # $ 01
1502 BRK
```

Das sollen uns die Register zunächst mal im Ausgangszustand zeigen. Nach G 1500 werden sie angezeigt:

```
AC XR YR N V - BDI ZC
01 00 00 0 0 110 000
```

Im Akku steht jetzt die dort eingeladene 1. Nun wollen wir das X-Register laden mit 255 (also $FF). Dazu ändern wir das Programm:

```
1502 LDX # $ FF
1504 BRK
```

Nach erneutem G 1500 zeigen die Register:

```
AC XR YR N V - BDI ZC
01 FF 00 1 0 110 000
```

Im X-Register steht nun die Zahl $FF. Bei den Flaggen hat sich die N-Flagge (die negative Zahlen anzeigen soll) auf 1 geschaltet!

Nun wollen wir das X-Register über 255 hinauszählen. Wir verändern das Programm nochmal:

```
1504 INX
1505 BRK
```

Der Start mit G 1500 liefert uns die folgende Registeranzeige:

```
AC XR YR N V - BDI ZC
01 00 00 0 0 110 010
```

Wie erwartet, ist der Überlauf des X-Registers eingetreten: Es ist jetzt Null. Die N-Flagge hat ihren gewohnten Wert 0 wieder angenommen und die Z-Flagge, die uns anzeigt, ob die letzte Operation eine Null erzeugt hat, ist jetzt gesetzt. Bei weiterem Hochzählen verschwindet die Z-Flagge wieder:

```
1505 INX
1506 BRK
```

G 1500 liefert den Registerinhalt:

```
AC XR YR N V - BDI ZC
01 01 00 0 0 110 000
```

Das gleiche passiert bei Verwendung des Y-Registers als Zähler, wie Sie leicht durch Austauschen aller auf X bezogenen Befehle feststellen können. Sehr nett ist es, diesen Befehlsablauf einmal für den INC-Befehl auf die Speicherstelle $0400 (Bildschirmspeicher links oben) bezogen ablaufen zu lassen. Wenn man darauf achtet, daß kein Hochscrollen des Bildschirms eintritt, kann man das Ergebnis außer in den Registern auch noch als Zeichen auf dem Bildschirm verfolgen. Der Beginn der Befehlsequenz ist dann sinnvollerweise:

```
1500 LDA # $ FF
1502 STA 0400
1505 BRK
```

Im folgenden setzt man dann anstelle von INX immer INC 0400 ein.

Was passiert beim Herunterzählen unter Null? Sie können das mit der gezeigten Befehlskette verfolgen, indem Sie immer statt INX jetzt DEX setzen und die Register nicht mit $FF, sondern mit 01 laden. Es zeigt sich, daß beim Herabzählen nach der Null wieder 255 (=$FF) im Register zu finden ist. Die Reaktion der N- und der Z-Flagge auf den jeweiligen Registerinhalt ist die gleiche wie beim Hochzählen.

13. Noch ein alchimistischer Zahlentrick: BCD

Es ist uns nun deutlich, daß diese sechs Befehle die N-Flagge und die Z-Flagge beeinflussen können. Diese Tatsache wird später noch eine große Rolle spielen, wenn es um die bereits erwähnte Schleifenkontrolle geht.

Die Assembler-Alchimisten haben noch viel mehr Arten der Zahlen- und Zeichendarstellung auf Lager. Eine davon ist die Codierung als BCD-Zahlen. BCD kommt vom englischen »binary coded dezimal«, was bedeutet: Binär codierte Dezimalzahlen.

Zwischendurch möchte ich noch eine Bemerkung loswerden, die Sie als Trost auffassen sollen: Auch wenn wir später andere Zahlendarstellungen kennenlernen werden, es wird nicht so schwierig! Sogar so komplette Idioten wie Computer verstehen das, obwohl man ihnen alles haarklein vorkauen muß.

Wenden wir uns nun wieder den einfachen BCD-Zahlen zu. Alle Zahlen von 0 bis 9 lassen sich binär mit nur 4 Bit ausdrücken:

Binär	Dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Die weiteren Werte 1010 bis 1111 werden in der BCD-Codierung nicht benutzt. Liegt nun eine Dezimalzahl (zum Beispiel 12) vor, dann wird jede Stelle dieser Zahl (also die 1 und die 2) getrennt binär codiert. In unserem Beispiel mit der 12 wäre das dann 0001 für die 1 und 0010 für die 2. Somit ist die 12 im BCD-Code 0001 0010. Jede Ziffer erhält so ihr Nibble. Eine Zahl im BCD-Format hat deswegen keine feste Anzahl von Bytes, sondern die Byte-Zahl hängt von der Anzahl der Stellen ab. Die Zahl 1984 beispielsweise braucht 2 Byte: 0001 1001 1000 0100.

Schwierig gestaltet sich das Rechnen mit diesen Zahlen wegen der sechs unbenutzten Codes. Aber auch da habe ich einen Trost für Sie: Wir werden damit nicht rechnen. Wozu das Ganze dann, werden Sie sich fragen? Der Grund für das alles ist, daß BCD-Zahlen im Gegensatz zu den Zahlen mit festem Format (die sonst verwendet werden) so eingegeben und verarbeitet werden können, wie sie vorliegen. Das ist im kaufmännischen Bereich manchmal notwendig, wo eben 1000mal 0,1 Pfennige 1 Mark ergeben und Fehler unzulässig sind. Sollten Sie also vor dem Problem stehen, mit BCD-Zahlen rechnen zu müssen, grämen Sie sich nicht: Unser Prozessor kennt den Dezimalmodus. Er ist dann eingeschaltet, wenn die Dezimal-Flagge auf 1 gesetzt ist.

Damit sollen Sie dann auch noch gleich zwei neue Befehle kennenlernen: SED und CLD. Der erstere hat nichts mit Parteien zu tun, sondern ist die Abkürzung für »SEt Dezimal-flag«, also setze die Dezimalflagge. So schalten Sie den Dezimal-Modus ein. Wie Sie sicher schon messerscharf geschlossen haben, heißt CLD »CLear Dezimalflag«, also setze die Dezimalflagge auf Null, wodurch dieser Modus wieder auszuschalten ist.

Wichtig! Wenn Sie argwöhnen, daß in einem Programm irgendwann mal die Dezimal-Flagge gesetzt sein könnte, dann gehen Sie auf Nummer Sicher und schieben immer vor eine Rechenoperation, die nicht im Dezimalmodus laufen soll, ein CLD.

Beide Befehle sind 1-Byte-Befehle mit implizierter Adressierung. Sie beeinflussen lediglich die Dezimalflagge.

Wie schon betont: Der Computer ist strohdumm. Er kann nicht einmal auf normale Weise voneinander abziehen! Deswegen geht er den komplizierten Weg: Er addiert eine negative Zahl. Nur: Wie sehen negative Binärzahlen aus?

Wir werden diese Frage in drei Etappen beantworten.
a) Man könnte eine Flagge setzen, die 1 ist bei negativen und 0 bei positiven Zahlen. Bei einigen Fließkommazahlen wird das auch so gemacht. Hier aber setzt man die Flagge direkt in die Zahl ein: Bit 7 jeder Zahl ist jetzt ein Vorzeichenmerkmal. Wenn dieses Bit 0 ist, handelt es sich um eine positive, wenn es 1 ist, um eine negative Zahl. Auf diese Weise ist also +1 wie bisher 0000 0001, wohingegen —1 jetzt 1000 0001 hieße. Damit wird allerdings der Zahlenbereich, der durch ein Byte auszudrücken ist, verschoben. 255=binär 1111 1111 kann so nicht mehr verwendet werden. Die größte Zahl, die jetzt ausgedrückt werden kann, ist 0111 1111 = dezimal 127. Die kleinste Zahl ist dann 1111 1111 = —127. Probieren wir mal aus, wie sich damit rechnen läßt:

```
    +10        0000 1010
    —6         1000 0110
    ---------------------
    ergibt     1001 0000 = —16,
```

was offensichtlich falsch ist, denn nach Adam Riese sollte +4 herauskommen. So kann man also nicht rechnen!

Man nennt diese Art der Zahlendarstellung übrigens »signed binary«-Format, also in Deutsch: markierte Binärzahlen.

b) Der nächste Schritt ist das sogenannte Einerkomplement. Dabei tritt für die positiven Zahlen keine Änderung ein. Die negativen entstehen aus den positiven durch Komplementbildung. Jedes Bit der positiven Zahl wird in sein Gegenteil verkehrt, wie es das folgende Beispiel zeigen soll:

```
    0000 1100   ist +12,
  dann ist das Einerkomplement:
    1111 0011   = —12.
```

Interessanterweise taucht hier auch wieder das Merkmal der »signed binary«-Zahlen auf: die 1 in Bit 7 bei negativen Zahlen. Beschränkt man sich auf den Zahlenbereich, der für die »signed binary«-Zahlen gültig war, dann hätten wir jetzt beide Darstellungsweisen miteinander vereint. Nun müssen wir natürlich noch feststellen, ob man so auch rechnen kann.

```
    +8              0000 1000
    —6              1111 1001
  in Einerkomplementdarstellung
  ---------------------------
    ergibt       (1) 0000 0001
```

was 1 mit einem Übertrag ergäbe, jedenfalls nicht 2, wie es sich gehört. Also ist auch die Einerkomplementdarstellung noch nicht das Gelbe vom Ei.

c) Ich will Sie nicht länger auf die Folter spannen: Wenn man zum Einerkomplement einer Zahl noch 1 dazuzählt, erhält man das Zweierkomplement. Und genau so werden negative Zahlen in unserem Computer gehandhabt. Die positiven Zahlen bleiben unverändert. Von den negativen bildet man das Zweierkomplement wie zum Beispiel hier mit der Zahl —12:

```
    12    0000 1100   normale Binärdarstellung
    —12   1111 0011   Einerkomplement
    +1    0000 0001   addieren
    ---------------
    —12   1111 0100   Zweierkomplement
```

Jetzt wollen wir auch diese Zahlenart ausgiebig testen: Wir rechnen nochmal 8-6:

```
    +8    0000 1000
    —6    1111 1010   das ist —6 in der
                      Zweierkomplementdarstellung.
    ---------------
  ergibt
         (1) 0000 0010
```

also 2 mit einem Übertrag, der ignoriert wird. Das Ergebnis ist richtig. Wenn bei einer solchen Rechnung eine negative Zahl herauskommt, ist sie nicht leicht zu erkennen. In solchen Fällen kehrt man das Vorzeichen um, indem man das Zweierkomplement berechnet. Das machen wir mal am Beispiel 5—6:

```
    +5    0000 0101
    —6    1111 1010   das ist wieder unser Zweier-
                      komplement von 6, also —6
    ---------------
    ergibt 1111 1111
```

das ist —1 in der Zweierkomplementdarstellung. Zur Kontrolle nun die Vorzeichenumkehr durch Umrechnen ins Zweierkomplement:

```
    Einerkomplement davon  0000 0000
    plus 1                 0000 0001
    --------------------------------
    ergibt                 0000 0001
```

also wie erwartet +1.

Auf diese Weise rechnet unser Computer mit negativen Zahlen. Negative ganze Zahlen speichert er im Zweierkomplement-Format. Auch wenn wir nun etwas vorgreifen müssen, wollen wir uns das ansehen. Dazu schalten Sie am besten erst einmal den Computer aus und laden dann den SMON beziehungsweise Ihren Assembler. Dann bauen wir ein kleines Basic-Programm:

```
10 A%=—12
20 END
```

14. Wie Variable im Speicher stehen

Noch nicht RUN eingeben! Zuerst schalten Sie den Maschinensprachmonitor ein und wir sehen uns das Programm so an, wie es im Speicher steht. Der Basic-Speicher des C 64 beginnt im Normalfall bei $0800. Wir geben also den Monitorbefehl M 0800.

Uns genügen schon die Speicherplätze bis $081C. Nun sehen wir das nackte Basic-Programm im Speicher.

In Bild 8 ist unser Speicherinhalt kommentiert zu sehen. Das Programm endet im Speicherplatz $0813. Das Kennzeichen für Programmende sind zwei aufeinanderfolgende

0800	00	0C	08	0A	00	41	25	B2
	080C Koppeladresse			000A Zeilennr.10		A	%	= Token
0808	AB	31	32	00	12	08	14	00
	— Token	1	2	Zeilen-ende	0812 Koppeladresse		0014 Zeilennr.20	
0810	80	00	00	00	FF	FF	FF	FF
	END Token	Zeilen-ende	Programm-ende		Leerer Speicher			

Bild 8. Der Monitor zeigt das nackte Programm im Speicher

Bytes mit dem Wort Null. Dahinter werden die Variablen abgelegt, sobald das Programm gestartet wird. Wir steigen aus dem Monitor durch X aus und starten das Programm mit RUN. Jetzt sehen wir noch mal in den Speicher. Bis $0813 hat sich nichts verändert. Danach aber ist jetzt in sieben Bytes die Variable A% abgelegt. Das zeigt Bild 9.

Zunächst einmal die Bytes $0814 und $0815: Hier wird der Variablenname und -typ angegeben. Der Typ ist aus den Bits 7 zu erkennen. Sind beide (wie hier) gleich 1, dann handelt es sich um eine Integervariable (also eine ganze

Speicherstelle	0814	0815	0816	0817	0818 bis 081A
Byte	1	2	3	4	5 — 7
Inhalt	C1 1100 0001	80 1000 0000	FF 1111 1111	F4 1111 0100	00 00 00 unbenutzt bei Integerzahlen
	Kennbits 7 für Integer 0100 0001 0000 0000 ≙ 65 Code für A		MSB von —12	LSB	
	Variablenname und -typ		Variablenwert		

Bild 9. So werden Integer-Variable aus Basic-Programmen vom C 64 im Speicher eingerichtet

Zahl). Läßt man die Kennbits außer acht, zeigt sich, daß in $0814 der Code für den Buchstaben A steht und $0815 nur den Wert 0 enthält. Nun zum Rest: Der C 64 legt Integers in nur 2 Byte ab – die restlichen 3 Byte $0818 bis $081A bleiben unbenutzt. Das ist auch dann der Fall, wenn danach noch weitere Variable kommen. Es bringt also keine Speicherersparnis, wenn man mit Ganzzahlvariablen arbeitet!

In $0817 steht $F4, welches binär ausgedrückt 1111 0100 ist. Das kennen wir noch von weiter oben als die —12 im Zweierkomplement-Format. Woher kommt $FF in Speicherzelle $0816? Wie gesagt, die Integers werden in 2 Byte gespeichert, und wenn wir —12 in 16 Bit ausdrücken, dann sieht das so aus:

+12	0000 0000 0000 1100
Einerkomplement:	1111 1111 1111 0011
plus 1	0000 0000 0000 0001
ergibt —12:	1111 1111 1111 0100
	MSB =$FF, LSB =$F4

als 16-Bit-Zweierkomplement.

Die größte positive ganze Zahl, die man in 2 Byte ausdrücken kann, ist 32767, was binär

0111 1111 1111 1111

ergibt. Die kleinste ist

1000 0000 0000 0000

also —32768. Das ist der Grund dafür, daß der C 64 Integers größer als 32767 oder kleiner als —32768 dankend mit ILLEGAL QUANTITY ERROR ablehnt, wenn sie als Argument verwendet werden.

Damit will ich Sie erstmal von den Zahlenspielereien erlösen. Sie können die Art des Abziehens von Zahlen durch Addieren des Zweierkomplementes bis zum nächsten Mal an weiteren Beispielen üben. Wenn Sie das mit 16-Bit-Zahlen tun, werden Sie bald feststellen, daß noch nicht alles so funktioniert wie es sollte...

Wir können jetzt übrigens auch das Rätsel lösen, weshalb bei positiven Zahlen (zum Beispiel LDA #FF) die Negativ-Flagge auf 1 geht: Die Flagge wird immer dann gezückt, wenn eine Zahl auftritt, die in Bit 7 eine 1 aufweist. Ganz einfach, gell?

15. Ein wirkungsvolles Zweiglein: BNE

Vermutlich raucht Ihnen nach soviel Zahlensalat der Kopf. Deshalb sollen Sie zur Entspannung noch einen neuen Assembler-Befehl kennenlernen und auch gleich ein nützliches Programmbeispiel dazu.

BNE heißt »Branch if Not Equal zero«, was man übersetzen kann mit »verzweige, wenn ungleich Null«. Genauer gesagt: Es wird dann verzweigt – also zu einer angegebenen Adresse gesprungen –, wenn die Z-Flagge (die haben wir bei den INX,DEX...-Befehlen genauer untersucht) nicht gesetzt ist, also 0 zeigt. Sehen wir uns das mal an der nachfolgenden Verzögerungsschleife an, deren Flußdiagramm Bild 10 zeigt.

Das Programm dazu:

```
1500    LDX # $ FF
1502    LDY # $ FF
1504    DEY
1505    BNE $ 1504
```



```
1507    DEX
1508    BNE $ 1502
150A    BRK
```

Zunächst einmal werden das X- und das Y-Register als Zähler initialisiert (also mit einem Ausgangswert geladen). Mit dem vorhin behandelten Befehl DEY wird dann das Y-Register um 1 heruntergezählt, was jetzt $FE ergibt. Für die Nullflagge (Z) bedeutet das den Inhalt 0, denn es liegt kein Grund vor, sie zu setzen (also eine 1 dort anzuzeigen), weil noch keine Null aufgetreten ist. Bei der nachfolgenden Prüfung durch BNE wird also eine Verzweigung nach 1504 das Ergebnis sein, worauf das Y-Register weiter verringert und dann die Z-Flagge erneut geprüft wird und so weiter. Das geht so lange, bis nun wirklich endlich die Null im Y-Register erreicht ist. In diesem Fall zählt DEX nun das X-Register herunter und der nächste BNE-Befehl führt zum Sprung nach 1502, wo das Y-Register wieder auf $FF gesetzt wird. Auf diese Weise wird die äußere Schleife 255mal und die innere 65025mal durchlaufen.

Sie haben beim Eingeben des Programmes vermutlich etwas gestutzt, als der Assembler nach dem BNE 1504 als nächste Adresse statt dem erwarteten 1508 eine 1507 ausgegeben hat. Der Befehl sieht zwar wie ein 3-Byte-Befehl aus, ist aber nur ein 2-Byte-Befehl! Das liegt an der speziellen Art der Adressierung von solchen Branch-Anweisungen: Der sogenannten relativen Adressierung, die wir aber erst später mit den anderen Branch-Befehlen behandeln werden.

Wenn Sie das Programm mit G 1500 starten, werden Sie - obwohl alles in Maschinensprache schnell läuft - eine merkliche Verzögerung feststellen, bevor die Registeranzeige auftaucht. Noch längere Verzögerungen lassen sich ohne weiteres erreichen, indem man mehr Schleifen ineinanderschachtelt. Dabei verwendet man dann den DEC-Befehl.

In der Tabelle 2 sind auch die Zyklen angegeben, die die neu gelernten Befehle zur Abarbeitung benötigen. Mit solchen Angaben lassen sich recht genau definierte Zeiten einstellen, in denen der Computer nichts anderes tut als durch das Programm zu flitzen. Wozu das dient, braucht wohl kaum noch gesagt werden: Wenn Sie zum Beispiel einen Text auf dem Bildschirm lesen wollen, bevor das Programm weiterläuft oder wenn Sie mit Peripherie arbeiten, die langsamer als das Programm ist oder... Allerdings muß noch gesagt werden, daß es noch elegantere Methoden zur Verzögerungs-Programmierung gibt als das Lahmlegen des Computers, aber dazu kommen wir erst später.

Befehls-wort	Adressierung	Byte-anzahl	Code Hex	Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
INX	implizit	1	E8	232	2	N,Z
INY	implizit	1	C8	200	2	N,Z
INC	absolut	3	EE	238	6	N,Z
DEX	implizit	1	CA	202	2	N,Z
DEY	implizit	1	88	136	2	N,Z
DEC	absolut	3	CE	206	6	N,Z
SED	implizit	1	F8	248	2	1 → D
CLD	implizit	1	D8	216	2	0 → D
BNE	relativ	2	D0	208	2 +1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	—

Tabelle 2. Die neuen Befehle

Neun neue Befehle haben wir bisher kennengelernt und wir wissen nun, wie unser Computer ganze Zahlen (sogenannte Integers) speichert. Zur Erinnerung: Das geschieht im Zweierkomplement-Format. Das Bit 7 einer 8-Bit-Zahl dient dabei als Vorzeichen-Merkmal: Wenn es 0 ist, liegt eine positive Zahl vor, die genauso aussieht, wie wir bislang immer Binärzahlen kannten. Ist das Bit 7 aber eine 1, dann haben wir es mit einer negativen Zahl in der Zweierkomplement-Darstellung zu tun. Wenn wir - wie unser Computer - zur Verarbeitung ganzer Zahlen 16 Bits (also 2 Bytes) verwenden, dann ist eben Bit 15 anstelle von Bit 7 das Vorzeichenbit.

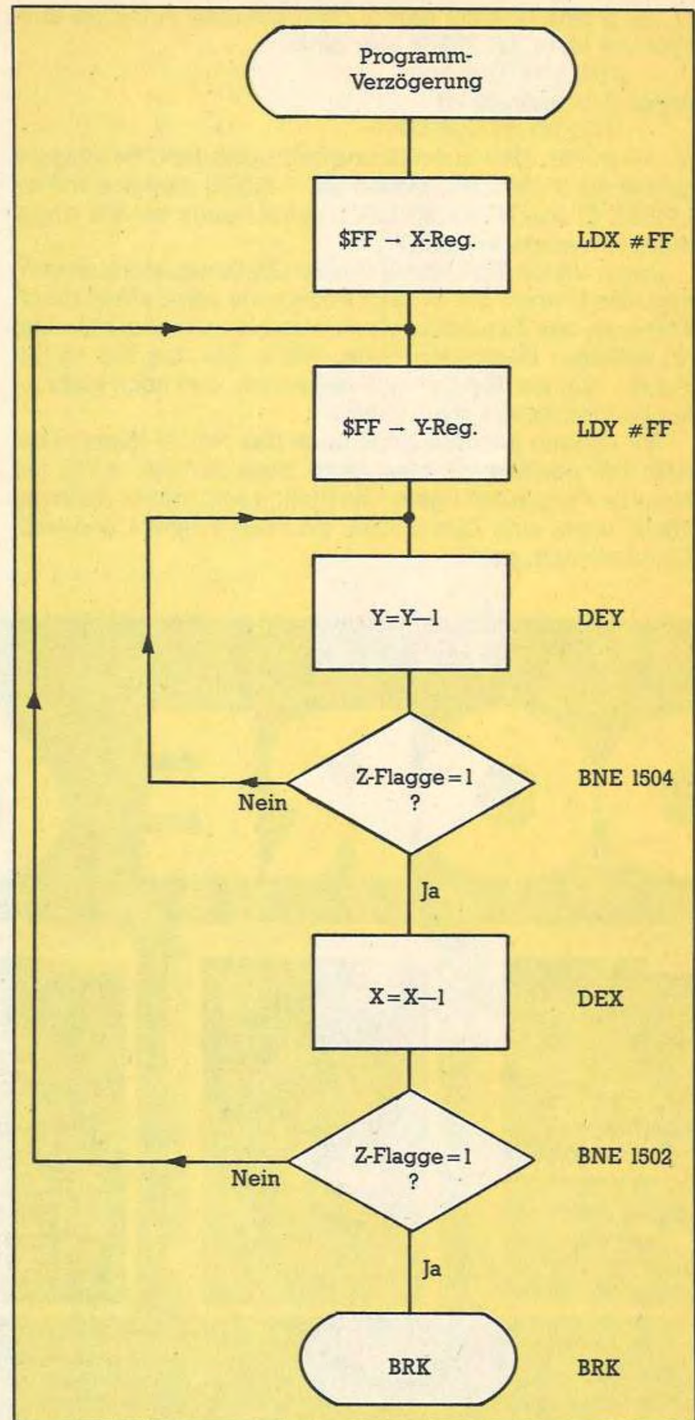


Bild 10. Flußdiagramm zur Verzögerungsschleife

16. Herr Carry und der V-Mann

Wenn Sie ein bißchen mit solchen Zahlen gerechnet haben, konnten Sie sicher feststellen, daß zwar oft das richtige Ergebnis herauskam - aber leider nicht immer.

Keine Angst, wir sind nicht ins Krimi- oder Agentenmilieu gewechselt! Wir haben es mit zwei Flaggen zu tun, der Carry- und der V-Flagge. »To carry« heißt auf deutsch etwa »tragen«. In der Registeranzeige ist diese Flagge immer mit C gekennzeichnet. Was wird denn hier getragen? Das ergründen wir am besten an einem Beispiel. Dazu rechnen wir mit normalen Binärzahlen (also ohne Rücksicht auf Vorzeichenbits). Wir zählen die Zahlen 128 und 130 zusammen:

```
  128       10000000
+ 130     + 10000010
--------------------
  258    (1)00000010
```

Das Ergebnis 258 ist richtig – auch in der Binärdarstellung – nur es paßt nicht mehr in 8 Bits. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen: In das Carry-Bit. Jedesmal also, wenn so ein Übertrag in einer Rechenoperation des C 64 stattfindet, zeigt die Carry-Flagge eine 1 (Bild 11).

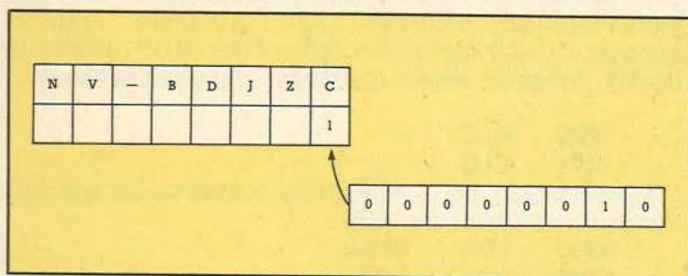


Bild 11. Das Carry-Bit als Bit 8 einer Rechenoperation

Je nach Art der von uns programmierten Aufgabe können wir nun dieses Carry-Bit weiterverarbeiten. Es gibt Situationen, in denen man es einfach ignorieren darf (dazu kommen wir gleich noch) oder aber solche, wo man es weiter in der Rechnung verwendet. Schließlich kann es auch noch einen Fehler anzeigen: Dann nämlich, wenn das größte zulässige Ergebnis 1111 1111 sein darf. Natürlich kann das Carry-Bit auch gesetzt werden, wenn man in der Zweierkomplementform rechnet. Die Verhältnisse sind dann aber für ein leicht überschaubares Beispiel des Übertrages zu verwickelt, wie Sie gleich sehen werden.

Wenn wir nämlich mit den Zweierkomplement-Zahlen rechnen, dann interessieren uns auch Fälle wie bei der Addition von 64 und 66:

```
    64       01000000
+   66     + 01000010
---------------------
 (-126)      10000010
```

Das ist offensichtlich falsch. Bei der Addition ist durch das Zusammenzählen der Bits 6 plötzlich Bit 7 gesetzt worden. Da wir es aber mit einer Zweierkomplementzahl zu tun haben, bei der dieses Bit 7 eine negative Zahl anzeigt, folgt ein Fehler. Es ist also von Bedeutung, so einen Überlauf (englisch: 'overflow') erkennen zu können um eine entsprechende programmtechnische Reaktion zu starten. Es wird die Überlauf-Flagge V auf 1 gesetzt. Leider ist die Sache aber nicht so einfach, daß sie immer gesetzt würde, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Gesetzt wird diese V-Flagge nur in folgenden zwei Fällen:

1) Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (wie beim Carry)

2) Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag.

Merken kann man sich das am besten so: Immer dann, wenn gewissermaßen das Vorzeichenbit 7 »versehentlich« verändert wurde, wird die V-Flagge auf 1 gesetzt. Das ist ein harter Brocken! Wir sind es ja gewohnt, daß wir uns um diese Dinge beim Computer eigentlich gar nicht mehr kümmern müssen. Außerdem würde das ja erfordern daß man sich bei allen Operationen vorher überlegen muß, welche Fehler also durch »versehentliches« Vorzeichenändern passieren können! Genauso ist es – in der Programmierpraxis wird Ihnen aber das ganze Problem nicht mehr so groß vorkommen. Wir wollen uns dieses Zusammenspiel der Überträge von Bit 6 nach Bit 7 und von Bit 7 nach Bit 8 (also in Carry-Bit) noch anhand einiger Beispiele klarer machen.

Im obigen Beispiel der Addition von 64 und 66 haben wir einen Fall schon behandelt: Es fand ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag in Carry-Bit. Deswegen wurde dann auch die V-Flagge gesetzt. Das Problem läßt sich hier ganz einfach lösen zum Beispiel durch Verwendung von 16-Bit-Zahlen:

```
   64      0000000001000000
+  66    + 0000000001000010
---------------------------
  130      0000000010000010
```

Bei 16-Bit-Zahlen ist ja Bit 15 das Vorzeichenbit, welches hier keine Änderung erfährt.

Der andere Fall tritt auf bei der Addition von zwei negativen Zahlen wie –125 und –64:

```
 -125       10000011
 - 64       11000000
--------------------
 (+67)   (1)01000011
```

Auch das ist offensichtlich falsch: Es hat wieder »versehentlich« ein Vorzeichenwechsel stattgefunden. Dies ist also der Fall, wo zwar ein Übertrag ins Carry-Bit stattfand aber kein Übertrag von Bit 6 nach Bit 7. Auch dieses Problem läßt sich durch Verwendung von 16-Bit-Zahlen lösen. Eine kleine Trainingsaufgabe für Sie!

Man kann also sagen: Immer dann, wenn bei 8-Bit-Rechnungen der mittels Zweierkomplementzahlen darstellbare Bereich (127 bis –128) über- oder unterschritten wird, fuhrwerkt man im Vorzeichen-Bit herum und verfälscht das Ergebnis. Dann leuchtet wie eine rote Ampel die Überlauf-(V-)Flagge auf und sagt uns, daß wir besser in diesen Fällen mit 16-Bit-Zahlen arbeiten sollten.

Nun noch zum Ignorieren des Carry-Bits, das ich weiter oben erwähnt habe. Bei allen 8-Bit-Rechenoperationen mit Zweierkomplementzahlen kann das Carry-Bit vernachlässigt werden. Zwei Beispiele sollen das wieder illustrieren. Wir addieren +4 und -2:

```
   +4        11111100
+  -2      + 11111110
---------------------
   -6     (1)11111010
```

Das Carry-Bit wird außer acht gelassen. Anderes Beispiel: Wir addieren zwei negative Zahlen, -4 und -6:

```
   -4        11111010
+  -2      + 11111110
---------------------
   -6     (1)11111000
```

Auch hier kann man (sogar: muß man) das Carry-Bit vernachlässigen. Beide Ergebnisse sind richtig.

Nun wissen Sie alles über die Art, wie unser Rechner mit ganzen Zahlen arbeitet. Probieren Sie mal ein paar Aufgaben aus zur Übung.

Wir verlassen jetzt den Zahlendschungel und widmen uns der Praxis.

ADC ist der erste Arithmetik-Befehl des 6502, den wir kennenlernen. Er bedeutet »ADd with Carry«, also »addiere mit Carry-Bit«. An einem 8-Bit-Beispiel wollen wir uns das mal ansehen. ZAHL1 und ZAHL2 sollen addiert werden.

17. Der Computer rechnet: ADC, CLC

Beide sollen positive 8-Bit-Zahlen sein, die so klein sind, daß kein Überlauf zu erwarten ist. Die ZAHL1 wird in den Akku gegeben:

```
LDA #ZAHL1
```

Wenn wir nun den Befehl

```
ADC #ZAHL2
```

folgen lassen, sorgt die ALU (arithmetisch-logische Einheit, siehe Kapitel 5) dafür, daß beide Zahlen addiert werden und das Ergebnis im Akku erscheint. ZAHL1 ist dann vom Ergebnis überschrieben worden. An sich ist damit alles erledigt. Weil wir aber häufig wissen wollen, was denn nun bei der Addition herausgekommen ist, speichern wir den Akku-Inhalt noch irgendwo ab mittels »STA Speicherstelle«. Außerdem war da ja noch die Sache mit dem Carry-Bit. Wir haben oben festgestellt, daß bei einer 8-Bit-Addition kein Carry-Bit berücksichtigt werden soll. Nun gibt es aber eine ganze Menge von Vorgängen in einem Assembler-Programm, die das Carry-Bit beeinflussen. Man kann eigentlich vor einer Addition nie ganz sicher sein, ob es denn nun 1 oder 0 ist. Weil jedoch ADC auch das Carry-Bit mitaddiert, sollte man dafür sorgen, daß es vor dem Zusammenzählen wirklich gelöscht ist. Dazu gibt es den Befehl CLC, was die Abkürzung für »CLear Carry«, also »lösche Carry-Bit« ist. Sei ZAHL1=12 und ZAHL2=7, dann würde unser vollständiges 8-Bit-Additions-Prögrämmchen also lauten:

```
1200   CLC
1201   LDA    #$0C
1203   ADC    #$07
1205   STA    $1500
```

Sehen wir mal davon ab, daß dieses Programm natürlich unsinnig ist (das kann man ja im Kopf schneller rechnen!), dann erkennen wir: CLC ist ein 1-Byte-Befehl mit impliziter Adressierung, welcher sich nur auf die C-Flagge (also das Carry-Bit) auswirkt. ADC ist in der hier verwendeten Form ein 2-Byte-Befehl und liegt in der »unmittelbar« genannten Adressierung vor. Wie wir oben gesehen haben, kann ADC - je nach Art der Rechnung - auf einige Flaggen wirken: Da wären zunächst natürlich die V-Flagge und die C-Flagge. Dann aber kann beim Auftreten eines gesetzten Bit 7 auch die N-Flagge und beim Überschreiten von $FF eventuell auch die Z-Flagge verändert werden.

Viel interessanter wird unser Mini-Programm schon, wenn man anstelle von

```
1201   LDA    #$0C
```

jetzt die absolute Adressierung verwendet, zum Beispiel

```
1201   LDA    $1400
```

Weil das ein 3-Byte-Befehl ist, verschiebt sich natürlich der Rest des Programmes um 1 Byte. So kann man immerhin schon zu unterschiedlichen Inhalten von 1400 den gleichen Betrag addieren.

Am interessantesten allerdings ist die Tatsache, daß auch ADC absolut adressierbar ist. Wir können so zum Beispiel den Inhalt der Speicherzelle $1300 zum Inhalt der Zeile $1400 hinzuzählen und dann das Ergebnis in $1500 ablegen:

```
1200   CLC
1201   LDA    $1400
1204   ADC    $1300
1207   STA    $1500
```

Hier ist der ADC-Befehl dann 3 Byte lang geworden.

Vergessen Sie bitte nicht - das gilt vor allem für die nachfolgenden Rechenoperationen - dann, wenn die Wahrscheinlichkeit besteht, daß der Dezimal-Modus eingeschaltet ist (also die D-Flagge auf 1 gesetzt ist), noch den Befehl CLD vor solche Programme zu stellen.

Solche 8-Bit-Rechnungen kommen recht häufig vor: Wenn man in Schleifen nicht mit mehrfach wiederholten INX (beziehungsweise INY oder INC, DEX, DEY oder DEC) arbeiten will, addiert man eben immer die Sprungweite mittels ADC hinzu. Der Akku kann nicht als Zähler dienen, denn es gibt für ihn keinen Befehl, der dem INX und so weiter vergleichbar wäre, weswegen man ihn - sollte es nötig sein - mittels ADC hochzählt.

Häufiger und in der Praxis bedeutender sind 16-Bit-Rechnungen. Wie Sie sicher noch aus den vorangegangenen Kapiteln wissen, teilt man so eine 16-Bit-Zahl auf in 2 Byte (das LSB und das MSB). Nehmen wir für unser nachfolgendes Beispiel wieder an, daß die Zahlen so gebaut sind, daß kein Überlauf zu befürchten ist. ZAHL1 hätten wir vorher in die Speicherstellen $1300 (LSB) und $1301 (MSB) gepackt, ZAHL2 liegt in den Zellen $1400 (LSB) und $1401 (MSB). Zunächst wieder die Vorbereitungsmaßnahmen:

```
1200   CLD
1201   CLC
```

Dabei ist CLD nicht immer nötig, wie schon gesagt. Nun addieren wir zuerst die LSBs:

```
1202   LDA    $1300
1205   ADC    $1400
1208   STA    $1500
```

Ein Überlauf kann hier nicht stattgefunden haben, denn das Vorzeichenbit ist ja im MSB als Bit 15 enthalten, wohl aber kann ein Übertrag stattgefunden haben: Das Ergebnis könnte größer als 255 ($FF) gewesen sein. War das der Fall, dann ist jetzt eine 1 im Carry-Bit. Wir addieren nun die MSBs:

```
120B   LDA    $1301
120E   ADC    $1401
1211   STA    $1501
```

Egal, was im Carry-Bit stand: Es wurde jetzt hinzuaddiert. Das Ergebnis unserer Rechnung steht nun in $1500 (LSB) und $1501 (MSB). Sehen wir uns das Ganze noch mal am Zahlenbeispiel an. Wir addieren die Zahlen 2176 (binär: 0000 1000 1000 0000) und 1009 (binär: 0000 0011 1111 0001). Die Speicherinhalte sind dann:

```
$1300  1000 0000   LSB Zahl1
$1301  0000 1000   MSB
$1400  1111 0001   LSB Zahl2
$1401  0000 0011   MSB
```

Jetzt addieren wir die LSBs:

```
$1300          1000 0000
$1400          1111 0001
Carry                  0
------------------------
$1500          0111 0001
Carry:                 1
```

Nun folgt der zweite Teil der Addition mit den MSBs:

```
$1301          0000 1000
$1401          0000 0011
Carry:                 1
------------------------
$1501          0000 1100
```

Damit steht nun das Ergebnis 3185 (binär 0000 1100 0111 0001) säuberlich aufgeteilt in LSB (Speicher $1500) und MSB (Speicher $1501) fest. Das Carry-Bit steht

auch nach vollendeter Rechnung noch auf 1, so daß es vor erneuter Addition wieder mit CLC zu löschen ist.

Damit wäre alles über die Addition berichtet. Wie immer in Programmiererkreisen die Empfehlung: üben, üben,....

Wir wenden uns jetzt der gegenläufigen Operation zu: der Subtraktion.

18. Noch mehr Rechnen: SBC, SEC

Daß das Abziehen von Zahlen im Computer durch das Hinzuzählen des Zweierkomplementes geschieht, haben wir mit viel Gehirnschmalzverbrauch schon in vorangegangenen Abschnitten erfahren. Nun sollen Sie die dazu nötigen Befehlsworte des Assemblers kennenlernen. Zunächst einmal ist da SBC. Das heißt »SuBtract with Carry« oder auf deutsch etwa »ziehe unter Berücksichtigung des Carry-Bits ab«. Ebenso wie bei der Addition mit ADC, wirkt das Argument des SBC-Befehls auf den Akku-Inhalt ein - wobei das Ergebnis im Akku landet, diesen also überschreibt. Komplizierter ist hier die Verwendung des Carry-Bits, worauf wir aber nicht detailliert eingehen wollen. (Wen es interessiert: Nachlesen in L.A. Leventhal, »6502 Programmieren in Assembler«, 3. Auflage, München 1983, Seite 3-100). Für uns soll einfach die nicht ganz korrekte Analogie zum »Borgen« bei der Subtraktion ausreichen. Für den Fall, daß ein solches Borgen eintreten muß, sollte auch das dazu nötige Carry-Bit vorhanden sein (also auf 1 gesetzt sein). Wie Sie sicherlich schon erraten haben, heißt SEC »SEt Carry«, also »setze das Carry-Bit« (auf 1).

Merke: Vor einer Addition immer Löschen des Carry-Bits mit CLC
Vor einer Subtraktion immer Setzen des Carry-Bits mit SEC!

Zwei Beispiele für die Subtraktion sollen das bisher Gesagte erläutern: Zunächst eine 8-Bit-Subtraktion von ZAHL1 (in Speicherzellle $1300) minus ZAHL2 (in Zelle $1400). Das Ergebnis wird nach $1500 geschrieben:

```
1200   CLD
1201   SEC
1202   LDA    $1300
1205   SBC    $1400
1208   STA    $1500
```

SBC kann - wie hier - absolut adressiert werden, aber auch unmittelbar (also zum Beispiel SBC #$40). Der Befehl ist dann im ersten Fall ein 3-, im anderen Fall ein 2-Byte-Befehl. SEC ist ebenso wie vorher schon CLC ein implizit adressierbarer 1-Byte-Befehl.

Das zweite Beispiel ist eine 16-Bit-Subtraktion. In den Speichern steht vor dem Aufruf dieser kleinen Routine:

```
$1300  ZAHL1  LSB
$1301  ZAHL1  MSB
$1400  ZAHL2  LSB
$1401  ZAHL2  MSB
```

Das Ergebnis soll nach $1500 (LSB) und $1501 (MSB) gebracht werden:

```
1200   CLD
1201   SEC
1202   LDA    $1300
1205   SBC    $1400
1208   STA    $1500
```

Jetzt sind die beiden LSBs voneinander abgezogen und die Differenz abgespeichert als LSB des Ergebnisses.

```
120B   LDA    $1301
120E   SBC    $1401
1211   STA    $1501
```

Damit ist die Aufgabe beendet. Auch die MSBs sind subtrahiert und das MSB des Ergebnisses steht in 1501.

SBC beeinflußt die gleichen Flaggen wie der Befehl ADC.

19. Ein Programmprojekt

Damit die kennengelernten Arithmetik-Befehle nicht so trocken auf weiter Flur stehen, wollen wir nun ein Programm entwickeln, aus dem zweierlei zu lernen ist:
1) Die Anwendung bisher gelernter Befehle und
2) ein häufig angewendetes Verfahren, Assembler-Programme in Basic-Programme einzubinden.

Besonders dieser zweite Aspekt scheint noch vielen Lesern unklar zu sein (das zeigen viele Zuschriften). Es gibt eine ganze Reihe von Möglichkeiten zum Einbau von Assembler-Routinen in Basic-Programme; diese werden wir alle nach und nach kennenlernen. Von Ihnen wurde der SYS-Befehl sicherlich schon häufig angewendet (zum Beispiel für SYS 58640 und vorherigem POKE214,Zeile und POKE211, Spalte zum Setzen des Cursors an die Stelle Zeile, Spalte). Damit haben Sie ein Maschinenprogramm aufgerufen, das im System unseres Computers schon enthalten ist. 58640 ist die Startadresse des Programmes und man kann diesen SYS-Befehl eigentlich wie eine Art »GOTO Maschinenprogramm-Startadresse« ansehen. Nichts hindert uns also, auf diese Weise eigene Assembler-Programme anzuspringen! Das Problem liegt nun nur noch darin, wie man Parameter, die unsere Maschinenroutine benötigt, übergeben kann. Eine offensichtliche - aber leider auch relativ langsame - Methode ist das POKEn der Werte im LSB/MSB-Format in die Speicherzellen, aus denen sie sich unser ML-Programm dann abholt. Wir wollen dieses Verfahren nun an einem Programmbeispiel verwenden.

Eine arithmetische Reihe werden viele von Ihnen schon kennen. Wenn man A als erstes Glied, D als Differenz und N als die Anzahl der Glieder bezeichnet, dann ist die Summe einer solchen Reihe:

$$S = A + (A + D) + (A + 2*D) + ... + (A + (N-1)*D)$$

Ein Beispiel ist die Summe der ersten zehn ganzen Zahlen:

$$S = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

Hier ist A = 1, D = 1 und N = 10. Daß die Summe S im Beispiel 55 ist, kann man schnell berechnen, was aber, wenn wir wesentlich mehr als nur zehn Glieder haben? Es gibt natürlich auch Formeln zur Berechnung von S. Aber eigentlich ist es ganz reizvoll, ohne solche Formeln den Computer die Summe bilden zu lassen. Wir bauen also ein Programm zur Berechnung der Summe der ersten N ganze Zahlen, wobei N frei gewählt werden kann. Das Ergebnis soll eine 16-Bit-Zahl sein, also nicht größer als 32767. Das beschränkt uns bei N auf Werte von 1 bis 255 (Warum, können Sie ja mal mit dem fertigen Programm ausprobieren). N benötigt also nur 1 Byte Speicherplatz und soll in $1300 abrufbar sein. A soll 1 sein ebenso wie D. Für eventuelle Programmänderungen ist es aber sinnvoll, A und D als 16-Bit-Zahlen aufzubewahren, und zwar in $1310/1311 (A in LSB/MSB-Format) und in $1320/1321 (D im gleichen Format). Das Ergebnis soll in $1400/1401 zu finden sein. Das Maschinenprogramm legen wir nach $1200.

Zuerst kümmern wir uns um das Basic-Aufrufprogramm (Listing 1):

Zu diesem Programm gibt es nur noch zu bemerken, daß die Zahlen bei POKE, PEEK oder SYS die Dezimalwerte unserer oben gewählten Adressen sind.

Nun endlich zum Assemblerprogramm. Sehen Sie sich dazu bitte das Flußdiagramm im Bild 12 an.

Wir bereiten den Ablauf vor, indem wir aus $1300 die Anzahl der Glieder ins X-Register laden und zur Vorbereitung der Addition das Carry-Bit löschen. Schalten Sie also bitte den SMON ein und tippen Sie A1200 <RETURN>. Es erscheint die Startadresse 1200. Jetzt können Sie Zeile für

```
10  REM **AUFRUF SUMME ARITHMETISCHE REIHE**
20  POKE5120,0:POKE5121,0:REM ERGEBNISSPEICHER AUF
    NULL
30  PRINTCHR$(147)CHR$(17)CHR$(17)
40  INPUT"ANZAHL DER GLIEDER N=";N
50  IFN<1 OR N>255 THEN PRINT CHR$(17)"1<=N
    <=255":GOTO40
60  POKE4864,N:REM EINSPEICHERN VON N
70  POKE4880,1:POKE4881,0:POKE4896,1:
    POKE4897,0:REM EINSPEICHERN VON A UND D
80  SYS4608:REM AUFRUF UNSERES MASCHINEN-
    PROGRAMMES
90  M=PEEK(5121):L=PEEK(5120):REM AUSLESEN DES
    ERGEBNISSES
100 E=256*M+L:PRINTCHR$(17)CHR$(17)
110 PRINT"DIE SUMME DER ERSTEN "N" GANZEN ZAHLEN
    IST:":PRINTE
120 END
```

Listing 1. Ein Basic-Programm bereitet eine Assembler-Addition vor

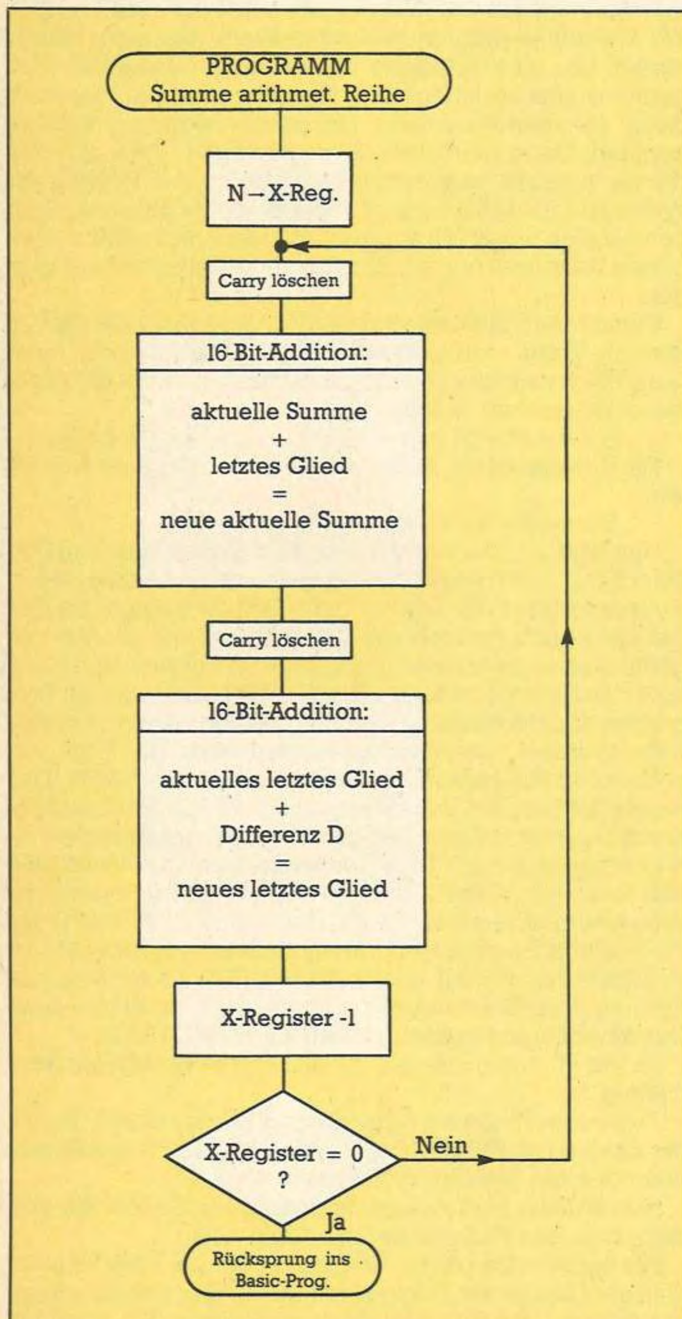
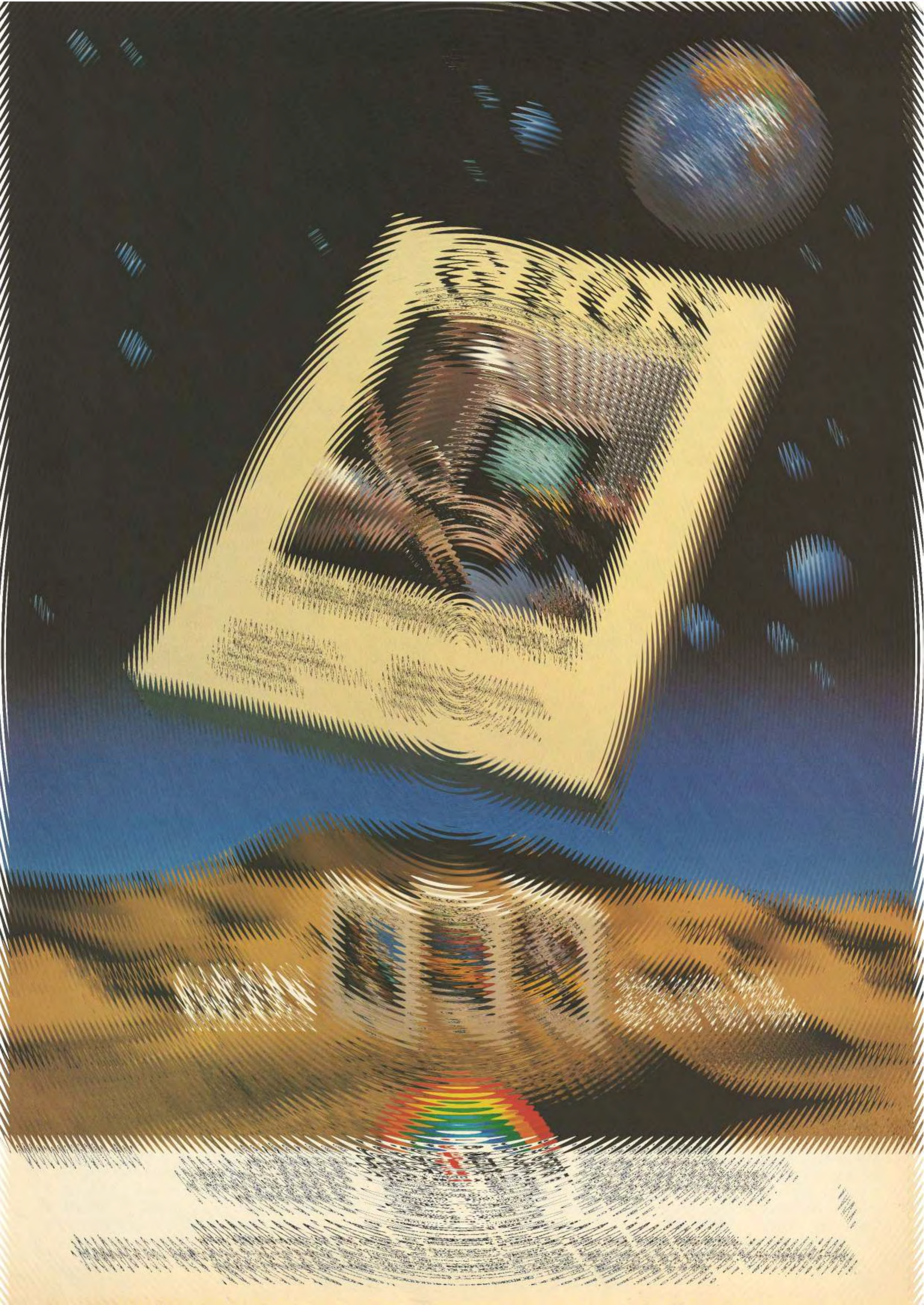


Bild 12. Flußdiagramm des Assemblerprogrammes »Summe einer arithmetischen Reihe«



Zeile das Assembler-Programm eingeben (nach jeder Zeile ein RETURN, das die nächste Zeilennummer erzeugt):

```
1200   LDX   $1300
1203   CLC
```

Die nächsten sechs Zeilen summieren jeweils das neueste Glied zur bis dahin erzeugten Summe. Jetzt zu Beginn ist $1400/1401 noch leer und in $1310/1311 steht noch das Anfangsglied A=1. Später mit Durchlaufen der Schleife, steht in $1400/1401 immer die bis dahin gebildete Summe und in $1310/1311 das letzte Glied der Reihe. Es handelt sich um die oben kennengelernte 16-Bit-Addition:

```
1204   LDA   $1400
1207   ADC   $1310
120A   STA   $1400
```

Das neue LSB ist berechnet und in $1400 geschrieben.

```
1200   LDA   $1401
1210   ADC   $1311
1213   STA   $1401
```

Das war nun noch das neue MSB. Als nächstes berechnen wir das momentan letzte Glied der Reihe durch Addieren von D zum alten letzten Glied. Das entspricht dem Basic-Befehl A=A+D in einer Schleife. Dies ist eine neue 16-Bit-Addition, weshalb wir wieder CLC vorgeben müssen:

```
1216   CLC
1217   LDA   1310
121A   ADC   $1320
121D   STA   $1310
```

Das war wieder das LSB. Nun zum MSB:

```
1220   LDA   1311
1223   ADC   1321
1226   STA   1311
```

Wir zählen nun das X-Register um 1 herunter und prüfen, ob es schon Null geworden ist, ob also schon alle N-Glieder summiert worden sind:

```
1229   DEX
122A   BNE   $1203
```

Wenn noch nicht alle Glieder berechnet und summiert sind, kehren wir an den Schleifenanfang zurück. Ansonsten springen wir zurück ins Basic-Aufrufprogramm:

```
122C   RTS
```

Wenn Sie beide Programme eingetippt haben, dann speichern Sie sie vorsichtshalber (das Assemblerprogramm mit dem S-Befehl des SMON). Beim neuen Einladen brauchen Sie den SMON nicht mehr. Nach dem Laden unseres Maschinenprogrammes (mit ,8,1 bei Diskette oder ,1,1 bei Kassette) geben Sie NEW <RETURN> ein, damit alle Zeiger vor dem Einladen des Basic-Programmes wieder auf Normalwerte gesetzt werden. Zwischen dem dann eingeladenen Basic-Programm und unserer Assembler-Routine ist genug Platz. Sollten Sie aber irgendwann mal das Basic-Programm vergrößern, schützen Sie bitte unseren Bereich ab $1200.

Unser Assembler-Beispiel ist so aufgebaut, daß auch A und D variabel gehalten sind. Sie müßten dann nur noch Eingabemöglichkeiten im Basic-Programm schaffen und anstelle der Werte 1 oder 0 in Zeile 70 die LSBs und MSBs der von Ihnen eingegebenen Größen A und D einPOKEn. Auf diese Weise sind dann beliebige ganzzahlige, arithmetische Reihen berechenbar, wie zum Beispiel S =7+10+13+16+... und so weiter. Das überlasse ich Ihrer Basic-Programmierfertigkeit. Nur eines noch: Sie müssen darauf achten, daß die Summe S nicht größer als 32767 wird. Ihrer Phantasie sind – wie immer in diesem Metier – keine Grenzen gesetzt. Sie könnten sich ja mal überlegen, wie man größere Summen zulassen kann (wer sagt denn, daß wir Zahlen immer nur in 2 Byte darstellen dürfen?). Oder Sie könnten sich überlegen, welches eindeutige Merkmal auftritt, sobald der Maximalwert überschritten wird (ein Tip: Lesen Sie doch mal den Abschnitt über die V-Flagge nach).

Der 6502 kennt acht bedingte Verzweigungen, von denen wir bisher BNE schon verwendet haben. Alle diese

20. Die Branch-Befehle

Branch-Befehle (von branch = verzweigen) prüfen Flaggen des Statusregisters.

BNE und BEQ beziehen sich auf die Z-Flagge, die anzeigt, ob im Verlauf der letzten Operation eine Null aufgetreten ist. Ist das der Fall, steht in der Z-Flagge eine 1. BNE verzweigt zur angegebenen Adresse, wenn in der Z-Flagge eine 0 enthalten ist. BEQ (»Branch if EQual zero« = »verzweige, wenn gleich Null«) tut das dann, wenn die Z-Flagge auf 1 gesetzt ist. Da muß man etwas aufpassen, daß man sich nicht vertut!

BCC und BCS haben ihre Aufmerksamkeit auf die C-Flagge, also das Carry-Bit gerichtet. BCC kommt vom englischen »Branch if Carry Clear«, was heißt: »verzweige, wenn das Carry-Bit gelöscht ist«. Ein gesetztes Carry-Bit (also Inhalt=1) veranlaßt BCS (»Branch if Carry Set« = verzweige, wenn das Carry-Bit gesetzt ist) zum Sprung an die angegebene Adresse.

Diese vier bedingten Verzweigungen sind an sich die bedeutsamsten und am häufigsten verwendeten Branch-Befehle. Man kann wohl getrost sagen, daß über 90% der von Programmierern verwendeten bedingten Sprünge damit absolviert werden. R. Mansfield warnt sogar ausdrücklich in seinem Buch »Machine language for beginners«, einem in den USA sehr verbreiteten Werk, vor der Verwendung der Befehle BPL und BMI!

Dafür liegt absolut kein einsehbarer Grund vor. Viele programmtechnischen Aufgabenstellungen lassen sich elegant und leicht mit BPL, BMI, BVS und BVC lösen. Man muß nur wissen, wie sie funktionieren und – da liegt vermutlich der Hund begraben – man muß auch die Art kennen, wie Zahlen vom Computer behandelt werden. Genau das aber wissen wir und deswegen sollten wir diese Kenntnis für uns auch nutzen. Also ohne Scheu heran an die verfehmten Befehle!

BMI und BPL (»Branch on MInus« = »verzweige, wenn negativ« und Branch on PLus« = »verzweige, wenn positiv«) hängen mit der Negativ-Flagge N zusammen. Das Rätsel dieser Flagge konnte in den vorangegangenen Folgen gelöst werden: Immer dann, wenn bei einer Operation eine Zahl auftrat, deren Bit 7 eine 1 war, wurde die N-Flagge auf 1 gesetzt. Wir wissen jetzt, daß dieses Bit bei 8-Bit-Zahlen das Vorzeichenbit ist. Bit 7 sagte uns bei einer 1, daß eine negative Zahl im Zweierkomplement-Format vorliegt oder aber überhaupt ein Speicherzelleninhalt vorhanden ist, der größer als 0111 1111 = 127 ist. BMI führt zum Sprung in diesem Fall, weil die N-Flagge auf 1 steht. Andernfalls führt BPL zur Verzweigung.

Ebenso einfach sind BVS und BVC zu erklären: Sie beziehen sich auf die V-Flagge, unsere rote Ampel, die Überlauf bei Rechenoperationen anzeigt. Kann es etwas bequemeres geben zur Behandlung solcher Fehlrechnungen als ein »Branch on oVerflow Set« = »verzweige, falls die Überlauf-Flagge gesetzt (=1) ist« mit BVS? Oder anders herum bei BVC »Branch on oVerflow Clear« = »verzweige bei freier Überlauf-Flagge«. Wenn man – wie Sie jetzt – weiß, unter welchen Umständen diese V-Flagge auf 1 gesetzt wird, sollte man ohne Skrupel BVS und BVC ausgiebig benutzen. Man könnte damit zum Beispiel programmieren, daß die Rechengenauigkeit automatisch von 16 Bit auf 24 oder 32 (oder wie es gerade beliebt) Bit gesteigert wird, ohne daß man sich bei jeder Programmaufgabe Gedanken über das

Befehls-wort	Adressierung	Byte-anzahl	Code hex	Code dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
ADC	unmittelbar	2	69	105	2	N,V,Z,C
	absolut	3	6D	109	4	
CLC	implizit	1	18	24	2	0 – C
SBC	unmittelbar	2	E9	233	2	N,V,Z,C
	absolut	3	ED	237	4	
SEC	implizit	1	38	56	2	1 – C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung
					+1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Tabelle 3. Die 11 neuen Befehle

größtmögliche Ergebnis machen muß. Dazu aber ein andermal mehr.

Alle hier vorgestellten Branch-Befehle sind ebenso wie BNE 2-Byte-Befehle, was an der speziellen Art der Adressierung liegt: der relativen Adressierung. Tabelle 3 zeigt eine Übersicht der neuen Befehle aus den letzten fünf Kapiteln.

21. Die relative Adressierung

Als wir den BNE-Befehl das erstemal verwendet haben, stellten wir fest, daß zum Beispiel BNE $1200 nicht — wie eigentlich zu erwarten war — ein 3-Byte-Befehl, sondern ein 2-Byte-Befehl ist. Damals mußten wir uns mit der Bemerkung zufriedengeben, es läge an der besonderen Art der Adressierung, nämlich der relativen Adressierung. Relativ bedeutet ja »bezogen auf etwas«. Wenn wir also beispielsweise BNE $1200 schreiben, liegt es nur an der Benutzerfreundlichkeit des SMON und vieler anderer Assembler, daß dieser die so geschriebene absolute Adresse $1200 in die richtige Form, nämlich die relative umrechnet. In Wahrheit verlangt der 6502 eine Angabe darüber, wie viele Bytes nach vorne oder hinten im Programm er zur weiteren Programmverarbeitung springen (verzweigen) soll. Es gilt nun also, zwei Fragen zu klären:
1. Relativ wozu wird gesprungen und
2. Wie berechnet sich die Angabe, um wie viele Bytes nach vorne oder hinten im Programm der Sprung vollzogen werden soll.

Zur Klärung verwenden wir ein hypothetisches Programmsegment mit einem Sprungbefehl und sehen uns das Disassemblerlisting an:

```
2000  AD 0030  LDA   $3000
2003  F0 05    BEQ   $200A
2005  A9 00    LDA   $#00
2007  8D 0030  STA   3000
200A  60       RTS
```

Dieses Programm-Teilchen lädt den Inhalt der Speicherstelle $3000 in den Akku, überprüft dann, ob dieser Inhalt Null ist und verzweigt beim Vorliegen der Null zum Rücksprung (RTS). Ist der Inhalt von $3000 nicht Null, dann wird $3000 auf Null gesetzt. $3000 könnte zum Beispiel eine Flagge sein.

Der Pfad, dem der Computer bei der Abarbeitung des Programms folgt, wird durch den Programmzähler vorbereitet. Dieser ist dann, wenn der BEQ-Befehl an der Reihe ist, schon einen Schritt weiter, nämlich im Programmzähler steht dann die Adresse 2005.

Relativ zu dieser Adresse hat dann der Sprung zu erfolgen. Zum Inhalt des Programmzählers muß also die Sprungweite (auch häufig Offset genannt) addiert werden. Soweit zur Frage 1.

Zur Klärung von Frage 2 listen wir uns mal Byte für Byte unser Programm auf:

Byte		Inhalt	Bedeutung
2000		AD	LDA
2001		00	LSB von $3000
2002		30	MSB von $3000
2003		F0	BEQ
2004		05	Offset
2005		A9	LDA #
2006	1	00	$00
2007	2	8D	STA
2008	3	00	LSB von $3000
2009	4	30	MSB von $3000
200A	5	60	RTS

Neben der Byte-Nummer ist noch die Entfernung zu 2005 geschrieben. Daraus ist deutlich zu erkennen, daß die Sprungweite, die zum Programmzähler addiert wird, 05 sein muß, wenn der Sprung zum RTS erfolgen soll. Für Vorwärts-Verzweigungen gilt also: Von der Adresse des Befehls an, der auf den Branch-Befehl folgt, zählt man die Byte-Anzahl bis zum Sprungziel. Das Ergebnis ist der Offset.

Nun gibt es genauso häufig Rückwärts-Sprünge. In den bisher gezeigten Programmen sind sie mehrmals aufgetreten. Wie berechnet man den Offset in diesen Fällen? Sehen wir uns wieder das Disassembler-Listing eines solchen Programmsegmentes an:

```
1000  A2 00  LDX $#00
1002  E8     INX
1003  D0FD   BNE $1002
1005  00     BRK
...
```

Dieses Programmchen tut nichts anderes, als das vorher auf Null gesetzte X-Register hochzuzählen, bis es über 255 läuft (dann tritt ja wieder 0 auf!). Solange der Inhalt des X-Registers ungleich Null ist, erfolgt ein Sprung zurück bis zur INX-Anweisung in Zeile 1002. Erst wenn die Null durch den Überlauf aufgetreten ist, endet das Programm mit einem BRK in Zeile 1005.

Wir wissen schon, daß der Programmzähler beim Verarbeiten des BNE-Befehls auf 1005 steht. Sehen wir uns auch dieses Programm Byte für Byte an:

Byte		Inhalt	Bedeutung
1000		A2	LDX #
1001		00	$00
1002	3	E8	INX
1003	2	D0	BNE
1004	1	FD	Offset
1005		00	BRK

Wieder ist neben der Byte-Nummer die Entfernung vom aktuellen Programmzählerstand angegeben. Wir müssen also vom Inhalt des Programmzählers 3 abziehen, um zum INX-Befehl in Byte 1002 zu gelangen. Das kennen wir aber schon aus den vergangenen Kapiteln: Wenn der Computer eine Zahl abzieht, dann addiert er das Zweierkomplement

dieser Zahl. Hier soll nun 3 subtrahiert werden. Wir berechnen das Zweierkomplement:

3 = 0000 0011 (binär)

Das Einerkomplement davon ist:

1111 1100

Dann wird eine 1 addiert

1111 1101

Dies ist das Zweierkomplement. In hexadezimal ausgedrückt heißt diese Zahl $FD und ist unser Offset. Für Rückwärts-Verzweigungen gilt also: Von der auf die Branch-Anweisung folgenden Speicherstelle an zählt man die Bytes zurück bis zum Sprungziel. Das Zweierkomplement der sich dadurch ergebenden Byte-Anzahl ist der Offset.

Das sieht reichlich kompliziert aus, aber zum einen haben Sie ja einen ganz freundlichen Assembler und nur in seltenen Notfällen müssen Sie den Offset berechnen. Zum anderen gibt es noch eine Faustregel, mit der man sich das Ganze vereinfachen kann. Die soll durch folgendes Schema erläutert werden:

Byte	Inhalt	Offset
...		
1995		F9
1996		FA
1997		FB
1998		FC
1999		FD
2000	BNE	FE
2001	Offset	FF
2002	Programm-zählerstand	
2003		01
2004		02
2005		03
..		

Bei Vorwärtssprüngen ist ohnehin alles klar: Bei einem Sprung nach Adresse 2005 müßte man in vorliegendem Fall einen Offset von 03 eingeben. Bei Rückwärts-Verzweigungen zählt man einfach von $FF an rückwärts bis zur Zieladresse. Eine Verzweigung nach 1996 würde im vorliegenden Fall also einen Offset von $FA erfordern.

Eine Einschränkung der relativen Adressierung können Sie nun auch sofort verstehen, wenn Sie an Zweierkomplementzahlen denken: Der Offset belegt ein Byte. Die größte positive Zahl in einem Byte ist

0111 1111 = +127 = $7F

und die kleinste negative Zahl ist

1000 0000 = —128 = $80

Es sind keine größeren Vorwärts-Verzweigungen als um 127 Byte möglich, weil in diesem Fall ein Offset größer als $7F, also mit einem Bit 7 gleich 1 nötig wäre, was aber wieder als negative Zweierkomplementzahl verstanden und einen Rückwärtssprung verursachen würde. Ähnliches gilt anders herum: Es ist kein weiterer Rücksprung als um 128 Byte möglich, weil das im Offset zum gelöschten Bit 7 führen würde, also zu einem Offset kleiner als $80, was wiederum anstelle des Rücksprunges eine Vorwärts-Verzweigung herbeiführen würde.

Man sollte beim Erstellen eines Assembler-Programmes darauf achten, daß man nie weitere Rückwärtssprünge als um 128, beziehungsweise Vorwärtssprünge um 127 Byte verlangt. Auch wenn man im Assembler gar nicht auf relative Adressierung Rücksicht nehmen muß, weil der Assembler sich mit den Absolut-Adressen begnügt, sollte man wissen, daß zum Beispiel folgende Zeile aufgrund dieser Einschränkung nicht möglich ist:

3000 BNE $1000

Die meisten Assembler reagieren auf solch eine Zeile mit einer Fehlermeldung (beim Hypra-Ass mit »Branch too far«) oder so wie der SMON, der klammheimlich die Programmstartadresse statt 1000 einsetzt. Aber es ist doch ärgerlich, wenn man auf dem Papier ein Programm fertig hat und erst beim Eintippen feststellt, daß der Computer das »so nicht haben will«.

22. Zeropage-Adressierung

Weil wir nun gerade mit der Adressierung so schön in Schwung sind, stelle ich Ihnen noch eine andere vor: Die Adressierung der Zeropage. Was ist die Zeropage? Auf deutsch heißt das Wort »Nullseite«. Am besten versteht man das, wenn man sich in Erinnerung ruft, wie Adressen in unserem Computer verwaltet werden. Da haben wir doch ein LSB (Least Significant Byte) und ein MSB (Most Significant Byte), zum Beispiel $1F 04 (mit 1F als MSB und 04 als LSB). Nun hat unser C 64 65535 Adressen von $0000 bis $FFFF. Bei den ersten 256 Adressen von $0000 bis $00FF ist das MSB $00. Man nennt so einen 256-Byte-Block eine Seite (engl. page). Weil hier für alle Adressen dieser ersten Seite des MSB Null ist, heißt sie Nullseite = Zeropage. Messerscharf werden Sie schließen, daß man die Seite mit

Befehlswort	Adressierung	Byteanzahl	Code Hex	Code Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	0-Page, abs.	2	A5	165	3	N,Z
LDX	0-Page, abs.	2	A6	166	3	N,Z
LDY	0-Page, abs.	2	A4	164	3	N,Z
STA	0-Page, abs.	2	85	133	3	—
STX	0-Page, abs.	2	86	134	3	—
STY	0-Page, abs.	2	84	132	3	—
INC	0-Page, abs.	2	E6	230	5	N,Z
DEC	0-Page, abs.	2	C6	198	5	N,Z
ADC	0-Page, abs.	2	65	101	3	N,V,Z,C
SBC	0-Page, abs.	2	E5	229	3	N,V,Z,C
CMP	unmittelbar	2	C9	201	2	
	absolut	3	CD	205	4	
	0-Page, abs.	2	C5	197	3	
CPX	unmittelbar	2	E0	224	2	N,Z,C
	absolut	3	EC	236	4	
	0-Page, abs.	2	E4	228	3	
CPY	unmittelbar	2	C0	192	2	
	absolut	3	CC	204	4	
	0-Page, abs.	2	C4	196	3	

Tabelle 4. Kenndaten der neuen Befehle und Adressierungen

den MSBs $01 als erste Seite bezeichnet, die mit den MSBs $02 als zweite Seite und so weiter.

Wenn wir nun zum Beispiel den Akku mit dem Inhalt der Zeropageadresse $00FA laden wollen, dann könnten wir schreiben:

3000 LDA $00FA

Unser Mikroprozessor versteht uns aber auch, wenn wir nur schreiben:

3000 LDA $FA

Das ist sie, die Zeropage-Adressierung. Anstelle eines 3-Byte-Befehls ist das jetzt ein 2-Byte-Befehl, was Speicherplatz und vor allem Rechenzeit einspart. Auf diese Weise kann man von den bisher kennengelernten Befehlen folgende adressieren:

LDA, LDX, LDY, STA, STX, STY, INC, DEC, ADC und SBC

Sie können sich merken, daß man (bis auf zwei Ausnahmen, die wir noch kennenlernen werden) alle absolut

64'er SONDERHEFT

PROGRAMM-SERVICE

Direkt bestellen statt abtippen!

Die aktuelle Diskette zum Heft:

64'er Sonderheft 35: Assembler

Hypra-Ass:
Ein Makro-Assembler der Spitzenklasse. Er erlaubt es, Maschinensprache-Programme ähnlich komfortabel wie in Basic zu schreiben. Durch Makros – dies sind immer wieder benötigte Unterprogramme, die über ihren Namen aufgerufen werden – und bedingte Assemblierung ist große Übersichtlichkeit auch bei langen Programmen gewährleistet. Ein weiterer Vorteil ist der Editor des Hypra-Ass, dessen formatierende LIST-Routine größtmögliche Übersicht am Bildschirm gewährleistet.

Reass:
Quasi die Umkehrung des Hypra-Ass und mindestens genauso wichtig ist dieser Reassembler. Sie haben beispielsweise ein Maschinenprogramm bekommen, das Ihnen gut gefällt, bis eben auf einige Kleinigkeiten, die Sie ändern wollen. Doch wie? Im Dickicht des reinen Maschinencodes findet sich niemand zurecht und ein Quellcode ist nicht vorhanden. Hier hilft der Reass: Er macht aus dem Maschinenprogramm wieder gut lesbaren, strukturierten Quellcode, der mit Labels und Zeilennummern versehen direkt mit dem Hypra-Ass bearbeitet werden kann.

SMON:
Dieser leistungsfähige Speichermonitor erlaubt Ihnen, bis in die tiefsten Tiefen Ihres Computers vorzudringen. Mit dem SMON lassen sich die Prozessorregister anzeigen und beeinflussen, Speicherbereiche anzeigen, vergleichen, verschieben und – und – und. Integriertes Disassemblieren erlaubt das Entschlüsseln von Maschinencode. Mit dem Miniassembler können Sie »mal eben« ein kleines Programm eingeben. Der Trumpf des SMON ist der ebenfalls integrierte Diskettenmonitor, der Ihnen zusätzlich volle Kontrolle über die Floppystation gibt.

Kurse:
Zwei komplette und leichtverständliche Kurse bringen selbst Assembler-Einsteiger auf das Profi-Niveau. Alle Programm-Routinen, die in den Kursen ausführlich erläutert werden, finden Sie ebenfalls auf dieser Programmservice-Diskette.
Weiterhin befinden sich auf der Diskette alle Programme, die im Inhaltsverzeichnis des 64'er-Sonderhefts 34 mit einem Diskettensymbol gekennzeichnet sind.

Diskette für C64/C128
Bestell-Nr. 15835

DM 19,90* (sFr 17,–*/öS 199,–*)
* Unverbindliche Preisempfehlung

Wenn Sie Fragen zu diesen Programmen oder zu anderen Angeboten aus unserem Programm-Service haben, rufen Sie uns an:
Telefon (089) 4613-640





Zeitschriften · Bücher
Software · Schulung

Weitere Angebote auf der Rückseite!

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bestellungen im Ausland bitte an: SCHWEIZ: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656. ÖSTERREICH: Markt&Technik Verlag Gesellschaft m.b.H., Große Neugasse 28, A-1040 Wien, Telefon (0222) 5871393-0; Rudolf Lechner & Sohn, Heizwerkstraße 10, A-1232 Wien, Telefon (0222) 677526. Ueberreuter Media Verlagsges. mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (0222) 481543-0

64'er

PROGRAMMSERVICE

Sie suchen packende Spiele, hilfreiche Utilities und professionelle Anwendungen für Ihren Computer? Sie wünschen sich gute Software zu vernünftigen Preisen? Hier finden Sie beides! Unser stetig wachsendes Sortiment enthält interessante Listing-Software für alle gängigen Computertypen. Jeden Monat erweitert sich unser aktuelles Angebot um eine weitere interessante Programmsammlung für jeweils einen Computertyp. Wenn Sie Fragen zu den Programmen in unserem Angebot haben, rufen Sie uns an: Telefon (089) 4613-640

Bestellungen bitte *nur gegen Vorauskasse* an: Markt&Technik Verlag AG, Unternehmensbereich Buchverlag, Hans-Pinsel-Straße 2, D-8013 Haar, Telefon (089) 4613-0. Schweiz: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Telefon (042) 415656. Österreich: Markt&Technik Verlag Gesellschaft m.b.H., Große Neugasse 28, A-1040 Wien, Telefon (0222) 5871393-0, Microcomput-ique, E. Schiller, Fasangasse 24, A-1030 Wien, Telefon (0222) 785661; Bücherzentrum Meidling, Schönbrunner Straße 261, A-1120 Wien, Telefon (0222) 833196. Ueberreuter Media, Verlagsges. mbH (Großhandel), Laudongasse 29, A-1082 Wien, Telefon (0222) 481543-0. Bestellungen aus anderen Ländern bitte nur schriftlich an: Markt&Technik Verlag AG, Abt. Buchvertrieb, Hans-Pinsel-Straße 2, D-8013 Haar, und gegen Bezahlung der Rechnung im voraus.

Bitte verwenden Sie für Ihre Bestellung und Überweisung die abgedruckte Postgiro-Zahlkarte, oder senden Sie uns einen Verrechnungs-Scheck mit Ihrer Bestellung. Sie erleichtern uns die Auftragsabwicklung, und dafür berechnen wir Ihnen keine Versandkosten.

64'er Sonderheft 34: Aufbruch in die dritte Dimension!

Fantastische Perspektiven: Ein Grafikprogramm mit ganz neuen Leistungsmerkmalen. Das Konstruieren von perspektivischen Grafiken, wie etwa ganzen Straßenzügen, wird zum Vergnügen. **Ein Freezer für harte Fälle:** Auf Knopfdruck wird der gesamte Speicher des C64 auf Diskette gespeichert und bei Bedarf wieder geladen. Sie arbeiten an der gleichen Stelle weiter, als wäre nichts geschehen. Sicherheitskopien von kopiergeschützten Originaldisketten anfertigen, Spielstände bei hartnäckigen Games »einfrieren« und bei Bedarf wieder laden – all das sind Stärken des Freezers.
3D-Movie-Maker: Das Konstruieren von dreidimensionalen Körpern und deren fließende Bewegung am Bildschirm zeichnet dieses Programm aus. Nach der Eingabe der Koordinaten für den Körper und seine Bewegung erzeugen Sie faszinierende Trickfilme in 3D.
Perfekte Simulation: Selten zuvor wurde so anschaulich gezeigt, wie eine Braunsche Röhre – Urahn aller Bildschirme und Monitore – funktioniert. Per Tastendruck steuern Sie alle Parameter. Veränderungen des Elektronenstrahls werden in Echtzeit am Bildschirm angezeigt.
Kurvendiskussion perfekt: Ein Segen für alle, die sich mit Mathematik und vor allem der Kurvendiskussion beschäftigen. Nicht nur, daß jede Funktion am Bildschirm anschaulich dargestellt wird. Auch die Ableitungen, Nullstellen etc. werden sofort berechnet und ausgegeben.
Digital einfach: Ideal für den Einstieg in die Digital-Elektronik ist dieses Programm. Alle Grundfunktionen, wie AND-, OR-, EXOR-Gatter etc. werden am Bildschirm dargestellt und in ihren Funktionen simuliert. Weiterhin befinden sich auf der Diskette alle Programme, die im Inhaltsverzeichnis des 64'er-Sonderhefts 34 mit einem Diskettensymbol gekennzeichnet sind.
Diskette für C64/C128

Bestell-Nr. 15834 **DM 29,90*** sFr 24,90*/öS 299,–*

64'er Sonderheft 33: Tips, Tricks & Tools zum C64/C128

Titelgenerator: Wenn Sie schon immer professionelle Programmvorspänne für Ihre eigenen Programme verwenden wollten: Jetzt können Sie es. Ohne Programmierkenntnisse erstellen Sie faszinierende Vorspänne mit Musik, Laufschrift, eigenen Zeichensätzen, Grafiken und und und...
Hypra-Sprite: Ein Sprite-Editor der Spitzenklasse, der Funktionen bietet, die auf diesem Gebiet noch nicht da waren. Mit einer komfortablen Benutzeroberfläche lassen sich Sprites erzeugen, dehnen, stauchen, spiegeln etc. Das Erzeugen von Animationssequenzen und kleinen Filmen gehört zu den Leckerbissen des Programms.
Basic-Kontroll-System: Machen Sie Schluß mit Syntaxfehlern und unsauberem Programmierstil in Ihren Basic-Programmen! Das Basic-Kontroll-System spürt die häufigsten Programmfehler auf und hebt so das Qualitätsniveau Ihrer Software.
Alpha-Drummer: Der C64 wird zum Super-Schlagzeuger. 24 digitalisierte Drum-Sounds stehen Ihnen sofort zur Verfügung. Weitere können mit der im Programm integrierten Digitizer-Software von Ihnen erzeugt und bearbeitet werden. Stücke mit einer Länge von über 15 Minuten sind kein Problem.
Apfelsee: Eine ganz neue Variante der Fraktalprogramme: Entdecken Sie die bizarre Schönheit dreidimensionaler Fraktal-Landschaften. Weiterhin befinden sich auf der Diskette alle Programme, die im Inhaltsverzeichnis des 64'er-Sonderhefts 33 mit einem Diskettensymbol gekennzeichnet sind.
Diskette für C64/C128

Bestell-Nr. 15833 **DM 29,90*** sFr 24,90*/öS 299,–*

64'er Sonderheft 32: Floppy- und Drucker-Software

Top-Flop Plus: Der Super-Disketten-Monitor für den C128 arbeitet in seiner erweiterten Version nun auch mit der neuen Floppy 1581 zusammen. Das Anlegen von Unterdirectories, den sogenannten »Partitions«, das neue Diskettenformat der Floppy 1581, wird für jeden transparent.
Kopieren wie ein Weltmeister können Sie mit dem Kopierprogramm »Hexer« für den C128 und einer Speichererweiterung 1750. Sogar eine doppelseitige 1571-Diskette mit 360 Kbyte paßt in den Arbeitsspeicher und kann in einem Arbeitsgang beliebig oft dupliziert werden.
Expansion-Basic: Erwecken Sie Ihre Speichererweiterung zum Leben: Den Betrieb einer blitzschnellen RAM-Floppy, Programmierung echter Windows mit Wiederherstellung des Bildschirminhalts und komfortable VDC-Programmierung – das alles bietet Ihnen diese komfortable Basic-Erweiterung für den C128.
Hardcopies der Spitzenklasse: für Epson-Drucker, den Star NL-10 und MPS 801 bieten Druckroutinen, Druckertreiber für die verschiedensten Programme. So zum Beispiel für: **EGA**, das tolle Zeichenprogramm aus dem 64'er-Magazin, **Amica-Paint**, das Super-Malprogramm aus dem Sonderheft 27. Die Star-NL-10-»Megadriver« für **GEOS** sorgen dafür, daß Schwarz wirklich schwarz ist und Kreise auch als solche gedruckt werden. Weiterhin befinden sich alle Programme auf der Diskette, die mit einem Diskettensymbol im Inhaltsverzeichnis gekennzeichnet sind.
Diskette für C64/C128

Bestell-Nr. 15832 **DM 29,90*** sFr 24,90*/öS 299,–*

64'er Sonderheft 29: Programme, die jeder C128-Besitzer braucht

Mastertext 128: Die Super-Textverarbeitung für den 80-Zeichen-Modus mit eingebauter Rechtschreibprüfung. Komfort und Funktionsvielfalt werden bei diesem Programm großgeschrieben. Alle Standardbefehle der modernen Textverarbeitung, ein integrierter Taschenrechner und sogar der Datenaustausch per DFÜ sind enthalten. Als besonderen Leckerbissen bietet Mastertext 128 eine Rechtschreibprüfung, deren Wörterbuch beliebig erweiterbar ist. Tippfehler gehören damit der Vergangenheit an!
Der Hexer: Endlich ein leistungsstarkes Kopierprogramm für den C128. Kopieren Sie nach Herzenslust, der Hexer wird auch Ihre Disketten bezaubern. Neben ganzen Disketten sind mit diesem Programm auch einzelne Files zu kopieren. Der Bedienkomfort des Hexers ist kaum zu überbieten. Probleme mit den verschiedenen Versionen des C128 kennt der Hexer nicht, es stehen verschiedene Versionen »für alle Fälle« bereit. Besitzer des »Dolphin-DOS« können sich über eine Version freuen, die mit diesem Floppy-Beschleuniger zusammenarbeitet.
Unidat Pro: Der Wunsch jedes ernsthaften Computer-Anwenders ist eine leistungsfähige und komfortable Dateiverwaltung. Mit Unidat Pro wird dieser Wunsch Realität. Erstellen und verändern Sie eigene Datei-Masken. Hohe Zugriffsgeschwindigkeit auf Ihre Daten, die Unterstützung von Paßwörtern zum Datenschutz und eine Export-Funktion zeichnen diese Dateiverwaltung aus. Die Suche nach einem Datensatz erfolgt blitzschnell.
Mancomania: Spielen Sie gerne Wirtschaftsspiele? Wenn Ihnen diese Spielgattung gefällt, ist Mancomania das Richtige für Sie. Das Spielziel ist allerdings ein wenig anders als bei den üblichen Vertretern dieses Genres: Verschleudern Sie Ihr Vermögen, so schnell Sie können. Vertreiben Sie sich die Zeit im Spiel-Casino, kaufen Sie Aktien an der Börse, und wetten Sie beim Autorennen. Denken Sie daran, das Geld muß weg. Aber das ist leichter gesagt als getan, als Millionär hat man's halt schwer!
Diskette für C128

Bestell-Nr. 15829 **DM 29,90*** sFr 24,90*/öS 299,–*

64'er Sonderheft 28: Programme und Utilities zu GEOS

Geoterm: Erschließen Sie sich die Welt der Datenfernübertragung mit GEOS. Geoterm ist ein Terminalprogramm der Spitzenklasse. Alle Funktionen sind wie von GEOS gewohnt mit Maus und Pull-down-Menüs steuerbar. So leicht war DFÜ noch nie. Sie wollen Ihre Grafiken, die Sie im Hi-Eddi-, Koalapainter-, Doodle-Format etc. vorliegen haben, in Geo-Write, GeoPaint GeoFile verwenden? Kein Problem, der **Bitmap-Converter** macht's möglich. Das Programm arbeitet vollständig unter GEOS und speichert Ihre Grafiken im Format von GEOS-Foto-Scraps. Diese können mit nahezu jedem GEOS-Programm weiterverarbeitet werden. Ärger mit dem Drucker? Erstklassige Qualität erhalten Ihre Ausdrucke unter GEOS mit den verschiedensten Druckertreibern für den Star NL-10, Epson-Drucker und Kompatible und den Citizen 120D. Mit **Superprint V2.0** läßt sich zudem nahezu jeder störrische Drucker an GEOS anpassen.
GEOS-Icon-Editor und **GEOS-Pattern-Editor:** Zwei Programme, die in keiner GEOS-Programmsammlung fehlen dürfen. Sie erlauben es, eigene Programme mit Icons (Piktogrammen) zu versehen. Mit dem Pattern-Editor kann jeder seine eigenen Füllmuster für GeoPaint nach Wunsch definieren. Eine Seite der Diskette wird im GEOS-Format ausgeliefert. Alle GEOS-Programme sind ohne Zusatzaufwand unter GEOS sofort lauffähig.
Datec: Ein Datenverwaltungsprogramm der Superlative (kein GEOS-Programm!). Freie Dateneingabe- und Druckmasken (beispielsweise für Etiketten) sind definierbar. Umfangreiche Such- und Indexfunktionen sowie frei definierbare Zeichensätze (natürlich mit deutschen Umlauten) sind nur einige der Glanzpunkte dieses Programms.
Diskette für C64/C128

Bestell-Nr. 15828 **DM 29,90*** sFr 24,90*/öS 299,–*

64'er Sonderheft 27: Ein unglaubliches Multicolor-Mal- und -Zeichenprogramm

Amica-Paint: Dieses Programm bietet Funktionen, die man vorher nur dem Amiga zugetraut hatte: Amica-Paint dreht, kippt und spiegelt beliebige Bildausschnitte und berechnet selbständig Farbverläufe. Definition von Makros, eine eingebaute Diashow-Funktion und natürlich komfortable Maussteuerung sind nur einige wenige Features dieses absolut sensationellen Programms.
Schreibmaschine: Entlocken Sie Ihrem Drucker Lettern in einer Qualität und Schönheit, die Sie ihm nicht zugetraut hätten. Viele Schriftarten befinden sich aus Platzgründen nur auf Diskette.
Pic-Change: Darauf haben die Grafik-Fans schon lange gewartet: Ob HiRes oder Mulicolor – Pic-Change macht Schluß mit dem Wirrwarr verschiedener Grafikformate. Jedes übliche Grafikformat kann in jedes andere übersetzt werden.
Grafik 2001: Eine leistungsfähige Erweiterung zur Basic-Erweiterung »Grafik 2000« mit vielen neuen Befehlen. Grafik 2000 (aus dem Sonderheft 4) ist auf der Programmservice-Diskette ebenfalls enthalten. Weiterhin finden Sie alle Programme auf Diskette, die im Inhaltsverzeichnis mit einem Diskettensymbol gekennzeichnet sind.
Zwei Disketten für C64/C128

Bestell-Nr. 15827 **DM 34,90*** sFr 29,50*/öS 349,–*

* Unverbindliche Preisempfehlung

Übrigens: Mit den Gutscheinen aus dem »Super-Software-Scheckheft« für DM 149,– können Sie sechs Software-Disketten Ihrer Wahl aus dem Programm-Service-Angebot der Zeitschriften

PC Magazin — **Happy-Computer-Sonderheft** — **Computer persönlich**
PC Magazin Plus — **Amiga-Magazin** — **64'er-Magazin**
Happy-Computer — **Amiga-Sonderheft** — **64'er-Sonderheft**

bestellen – egal, ob diese DM 29,90 oder DM 34,90 kosten. Das Scheckheft können Sie per Verrechnungsscheck oder mit der eingehefteten Zahlkarte direkt beim Verlag bestellen.
Kennwort: Software-Scheckheft, Bestell-Nr. 39100.

adressierbaren Befehle auch Zeropage-absolut anwenden kann. Genauere Angaben über die Codes, die Ausführungszeiten und die Beeinflussung der Flaggen (letztere ist identisch mit der absoluten Adressierung) entnehmen Sie bitte der angefügten Tabelle 4.

Zum Thema Geschwindigkeit: Wenn Sie die benötigten Taktzyklen von absolut und von 0-absolut adressierten Befehlen in den Tabellen miteinander vergleichen, werden Sie jeweils einen Unterschied von einem Zyklus feststellen. Das mag Ihnen läppisch vorkommen. Bedenken Sie aber, daß Sie sehr häufig Schleifen programmieren müssen, die mehrere 100mal durchlaufen werden, die vielleicht als oft zu verwendende Unterprogramme dienen... Sie werden bald feststellen, daß da schnell beachtliche Zeitunterschiede auftreten können: Für zeitkritische Programme ist die Verwendung der Zeropage-Adressierung dringend geboten.

Dieser Tatsache waren sich leider auch die Schöpfer unseres Betriebssystems und des Basic-Interpreters voll bewußt. Die Zeropage ist nahezu randvoll mit Speicherstellen, in denen sich beide Programmkomplexe tummeln. Fast jede Kernel- und Interpreter-Routine notiert sich irgendwelche Werte auf der Seite Null. Das macht es uns als Assembler-Programmierer nicht gerade leicht, die Zeropageadressierung zu verwenden, wenn wir außerdem den Interpreter oder das Betriebssystem benutzen wollen. Es kann geradezu katastrophale Folgen haben, einige Zeropage-Adressen zu überschreiben. Andere werden ständig neu beschrieben durch das Betriebssystem oder den Interpreter, was unseren eigenen — vielleicht gerade in so einer Speicherzelle gelagerten — Zwischenwerten den Garaus machen würde. Man sollte sich also die ersten 256 Speicherstellen ganz genau ansehen, bevor man sie adressiert oder aber auf das Betriebssystem und den Basic-Interpreter verzichten. Ersteres erleichtern uns Tabellen der Speicherbelegung (zum Beispiel Babel, Krause, Dripke »Das Interface Age Systemhandbuch zum Commodore 64«, Interface Age Verlag, oder »Das Commodore 64 Buch, Band 4, Ein Leitfaden für Systemprogrammierer«, Markt und Technik Verlag) und auch die Serie von Dr. Helmut Hauck »Memory Map mit Wandervorschlägen«, die ab Ausgabe 11/84 im 64'er-Magazin erschien.

Ohne Hemmungen dürfen wir nur die Speicherstellen (jedenfalls beim C 64) $02 und $FB bis $FE nutzen, wenn GEOS nicht aktiv ist. Weil das doch recht mickrig ist, hat jeder Assembler-Programmierer spezielle Tips, welche Zellen er noch mit welchen Vorsichtsmaßnahmen benutzt. Wenn man bestimmte Routinen aus dem Betriebssystem oder dem Interpreter nicht aufruft, bleiben dazugehörige Zeropage-Adressen unbeeinflußt und sind dann für eigene Zwecke nutzbar. Manchmal ist es notwendig, den alten Zustand einer Adresse nach Beendigung eigener Programme wieder herzustellen, manchmal nicht. Interessant und viel beschrieben in allen möglichen Zeitschriften und Büchern ist die Möglichkeit, die Notizen, die sich das Betriebssystem oder der Interpreter auf der Zeropage macht, zu verändern. Im Prinzip schreibt man damit kleine Teile dieser Großprogramme um oder variiert Tabellenteile davon. Das geschieht im Rahmen der »Tricks« mit irgendwelchen POKEs mehr oder weniger blind, weshalb auch bevorzugt Abstürze des Computers dabei festzustellen sind. Warum Abstürze? Na, stellen Sie sich mal ein von Ihnen geschriebenes Programm vor — zum Beispiel das aus Kapitel 19 zur Berechnung der Summe einer arithmetischen Reihe — und POKEn Sie dann anstelle irgendeines Befehlscodes, der dorthin gehört, jetzt eine 0 (also ein BRK) hinein. Die Wirkung dürfte ähnlich sein. Wenn man allerdings die Funktion der betreffenden Speicherstelle genau kennt, lassen sich recht nützliche Änderungen hervorrufen, wie zum Beispiel die Schutz-POKEs für den Basic-Speicher durch Verändern der Adressen $33, $34, $37 und $38.

Wir werden im folgenden immer dann, wenn wir mit Zeropage-Adressierung arbeiten oder Routinen des Betriebssystems oder Interpreters untersuchen, spezielle Stellen der Nullseite kennenlernen.

Vorhin hatte ich noch angedeutet, daß man dann die Zeropage fast vollständig nutzen könne, wenn man auf den Basic-Interpreter und das Betriebssystem verzichtet. Das ist tatsächlich möglich. Nur wird man dann erstaunt feststellen, wieviel Arbeit uns die computer-interne Software abnimmt oder anders herum: Viele bislang selbstverständliche Dinge werden wir dann plötzlich selbst programmieren müssen, und das kann ein hartes Brot sein!

Als Beispiel für ein Programm, das nicht nur die Zeropageadressierung verwendet, sondern sogar selbst komplett in der Zeropage steht, werden wir uns die CHRGET-Routine ansehen. Eine Klasse von Befehlen, die dort angewendet wird, die Vergleichsbefehle, soll zuvor noch gezeigt werden.

23. Die Vergleichsbefehle: CMP, CPX, CPY

Vergleichen heißt in englischer Sprache »to compare«, woraus Sie unschwer erkennen können, woher die Bezeichnung CMP und die CPs in CPX beziehungsweise CPY kommen. Verglichen wird jeweils der Akku-Inhalt (bei CMP), der Inhalt des X- (bei CPX) oder des Y-Registers (bei CPY) mit Daten, die der Compare-Befehl adressiert. Einige Beispiele werden Ihnen das klarer machen:

```
CMP #$FF
```

vergleicht den Akku-Inhalt mit der Zahl $FF. Hier liegt die unmittelbare Adressierung vor, die ebenso für CPX und CPY verwendbar ist. Außerdem ist das dann ein 2-Byte-Befehl.

```
CPX $3000
```

vergleicht den Inhalt des X-Registers mit dem Inhalt der Speicherstelle $3000. Die absolute Adressierung ist also auch anwendbar (natürlich auch für CMP und CPY). Der Compare-Befehl besteht so aus 3 Byte.

```
CPY $A8
```

vergleicht den Inhalt des Y-Registers mit dem Inhalt der Zeropagestelle $A8. Diese soeben frisch gelernte Zeropage-Adressierung ist bei allen drei Vergleichsbefehlen möglich und macht aus ihnen 2-Byte-Befehle.

Für CPX und CPY sind das alle Möglichkeiten der Adressierung. CMP erlaubt weitere, die wir noch kennenlernen werden. Nun interessiert uns natürlich noch, wie das Vergleichsergebnis zu erhalten ist! Bei diesen Befehlen geschieht Merkwürdiges: Die Vergleichsdaten werden vom Inhalt des Akkus (beziehungsweise X- oder Y-Registers) abgezogen, aber: Weder wird dieser Inhalt noch werden die adressierten Daten verändert! Der Trick ist, daß drei Flaggen das Ergebnis anzeigen: Die Negativ-Flagge N, die Null-Flage Z und das Carry-Bit C. Diese Anzeige geschieht so:

1) Der Registerinhalt (Akku, X-, Y-Register) ist größer als die Vergleichsdaten:
Dann ist das Carry-Bit = 1, die N- und die Z-Flagge =0.

2) Der Registerinhalt ist gleich den Vergleichsdaten:
Dann sind Carry- und Z-Flagge = 1, die N-Flagge = 0.

3) Der Registerinhalt ist kleiner als die Vergleichsdaten:
Die N-Flagge ist dann =1, Carry- und Zero-Flagge sind 0.

Damit Sie die Übersicht behalten können, ist in Bild 13 das Ganze als Schema gezeigt.

Sie werden sich vermutlich schon denken können, wie der Hase weiterläuft: Mit den Verzweigungsbefehlen prü-

fen wir die Flaggen und springen die gewünschten weiteren Programm-Routinen an.

Die Kombination der Compare-Befehle mit den Verzweigungsoperationen wird Ihnen im weiteren Verlauf dieses Kurses noch ganz geläufig werden. Ein Beispiel sehen Sie nachher ebenfalls in der CHRGET-Routine. Leider muß ich Sie immer noch etwas vertrösten, denn mit Verstand begreifen läßt sich diese Routine nur dann, wenn man etwas mehr über die Codierung von Zeichen weiß. Deswegen werden wir uns nun noch mit dem ASCII-Code und dem Commodore-ASCII beschäftigen.

FLAGGE	Akku X Y } >DATEN	Akku X Y } =DATEN	Akku X Y } <DATEN
N	0 oder 1	0	1 oder 0
Z	0	1	0
C	1	1	0

Bild 13. Flaggen bei den Vergleichsbefehlen

24. Zeichencodierung mit dem ASCII- und dem Commodore-ASCII-Code

ASCII ist die Abkürzung von »American Standard Code for Information Interchange« und das heißt auf deutsch »amerikanischer Standard-Code zum Informations-Austausch«. Diese Zeichenverschlüsselungsart ist international als ISO-7-Bit-Code genormt, und es wäre wirklich nett, wenn alle sich daran halten würden. Tatsächlich aber finden wir zum Beispiel bei unserem C 64 eine Abart des Normcodes, den Commodore-ASCII-Code. Über die damit erzwungenen Umrechnungen können alle diejenigen Dramen erzählen, die zum erstenmal einen (Nicht-Commodore-)Drucker an ihr Gerät anschließen oder aber blauäugig in den Online-Betrieb mit anderen Computern eintreten wollten.

Sehen wir uns zunächst einmal den ASCII-Code an. Es handelt sich um einen 7-Bit-Code, das heißt 128 Zeichen können in nur 7 Bit untergebracht werden (0000 0000 bis 01111111). Das achte Bit dient bei manchen Operationen mit Computer-Peripherie als Paritäts-Bit. Bei dieser Gelegenheit soll auch gleich erklärt werden, was Parität in diesem Zusammenhang bedeutet. Werden Daten übertragen, muß immer mit Übermittlungsfehlern gerechnet werden. Das Paritätsbit dient dazu festzustellen, ob ein Byte korrekt angekommen ist. Bei der sogenannten geraden Parität zählt man die Einser im Byte zusammen und setzt Bit 7 auf 1 wenn sich eine ungerade Zahl ergibt. Mit dem Paritätsbit haben wir dann eine gerade Zahl. Ist die Quersumme des Byte schon gerade, bleibt Bit 7 eine Null. Ebensogut kann man die ungerade Parität verwenden, indem dann Bit 7 so gewählt wird, daß sich immer eine ungerade Zahl ergibt. Welche Art der Parität zur Anwendung kommt, ist Vereinbarungssache. Nehmen wir mal an, es sei gerade Parität gefordert und ein Byte mit der Information 00010110 soll übermittelt werden. Die Quersumme ist 3, also ungerade. Das Paritätsbit muß auf 1 gesetzt werden. Wir senden das Byte 10010110. Der Empfänger überprüft zunächst auf gerade Parität und verwendet dann nur die Bits 0 bis 6. Doppelfehler, die mittels des Paritätsverfahrens nicht festgestellt werden können, sind sehr selten. Leider kann auf diese Weise nur bemerkt werden, daß ein Übertragungsfehler aufgetreten sein muß, aber nicht welcher. Die Information muß dann neu angefordert werden.

Sehen wir uns nun den Commodore-ASCII-Code an. Durch die Einbindung der Grafikzeichen brauchen wir mehr als die 128 Kombinationen. Commodore benutzt deswegen einen 8-Bit-Code. Mit dem Basic-Befehl CHR$(x) können Sie sich alle 256 Möglichkeiten ansehen. Erschwerend kommt aber noch hinzu, daß wir nicht nur einen Zeichensatz, sondern deren vier zur Verfügung haben, die durch den jeweiligen Schreibmodus ansprechbar sind (Klein-/Großschriftmodus, Großschriftmodus, beide Modi mit Reverse-ON oder OFF). Im Zeichen-ROM liegen insgesamt 512 Muster abrufbereit. Zu diesen kommen beim CHR$-Befehl noch eine ganze Reihe von Steuerzeichen

lsn $	msn	$ bin. binär	0	1	2	3	4	5	6	7
			0000	0001	0010	0011	0100	0101	0110	0111
0	0	000	NUL	DLE	SP	0	@	P		p
			NULL	DLE	SP	0	@	P	CHR$(96)	CHR$(112)
1	0	001	SOH	DC1	!	1	A	Q	a	q
			SOH	DC1	!	1	A	Q	CHR$(97)	CHR$(113)
2	0	010	STX	DC2	"	2	B	R	b	r
			STX	DC2	"	2	B	R	CHR$(98)	CHR$(114)
3	0	011	ETX	DC3	#	3	C	S	c	s
			ETX	DC3	#	3	C	S	CHR$(99)	CHR$(115)
4	0	100	EOT	DC4	$	4	D	T	d	t
			EOT	DC4	$	4	D	T	CHR$(100)	CHR$(116)
5	0	101	ENQ	NAK	%	5	E	U	e	u
			ENQ	NAK	%	5	E	U	CHR$(101)	CHR$(117)
6	0	110	ACK	SYN	&	6	F	V	f	v
			ACK	SYN	&	6	F	V	CHR$(102)	CHR$(118)
7	0	111	BEL	ETB	'	7	G	W	g	w
			BEL	ETB	'	7	G	W	CHR$(103)	CHR$(119)
8	1	000	BS	CAN	(	8	H	X	h	x
			BS	CAN	(	8	H	X	CHR$(104)	CHR$(120)
9	1	001	HT	EM	)	9	I	Y	i	y
			HT	EM	)	9	I	Y	CHR$(105)	CHR$(121)
A	1	010	LF	SUB	*	:	J	Z	j	z
			LF	SUB	*	:	J	Z	CHR$(106)	CHR$(122)
B	1	011	VT	ESC	+	;	K	[	k	{
			VT	ESC	+	;	K	[	CHR$(107)	CHR$(123)
C	1	100	FF	FS	,	<	L	\	l	¦
			FF	FS	,	<	L	£	CHR$(108)	CHR$(124)
D	1	101	CR	GS	—	=	M	]	m	}
			CR	GS	—	=	M	]	CHR$(109)	CHR$(125)
E	1	110	SO	RS	·	>	N	↑	n	~
			SO	RS	·	>	N	↑	CHR$(110)	CHR$(126)
F	1	111	SI	US	/	?	O	—	o	DEL
			SI	US	/	?	O	←	CHR$(111)	CHR$(127)

Bild 14: ASCII-Code (jeweils oben) und Commodore-ASCII-Code (jeweils unten) (msn = most significant nibble; lsn = least significant nibble)

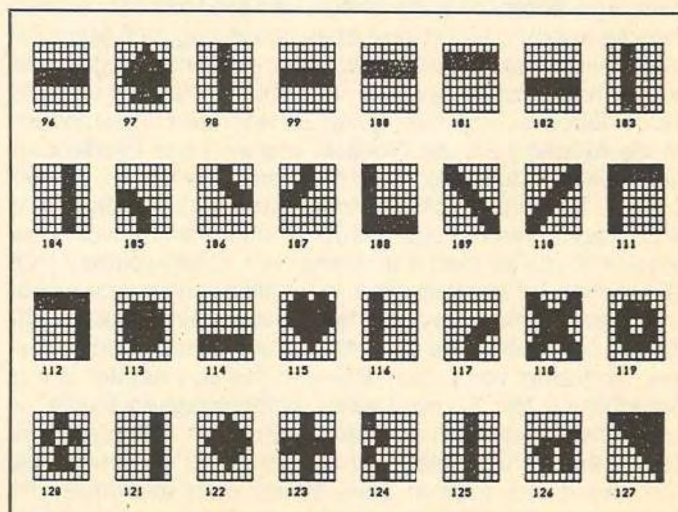


Bild 15. Grafikzeichen zu den entsprechenden CHR$-Codes

hinzu... die Verwirrung ist perfekt! Wir wollen an dieser Stelle keine Entwirrung vornehmen, sondern wir durchschlagen den Gordischen Knoten, indem wir nur die ersten 128 Zeichen mit den ASCII-Zeichen vergleichen. In Bild 14 und 15 finden Sie unsere Gegenüberstellung.

Einige Kombinationen dienen als Steuer-Codes. (Die Bedeutung der dabei verwendeten Abkürzungen sehen Sie in Bild 16.)

Nur ein Teil dieser Codes wird tatsächlich genutzt. Andere haben - je nach Gerät, an das sie gesandt werden - unterschiedliche Bedeutungen. Denken Sie dabei nur mal an die verschiedenen Betriebssysteme des Commodore-Druckers 1526, wo man bei dem einen mit CHR$(1), bei dem anderen mit CHR$(14) den Breitschrift-Modus anschaltet. Innerhalb unseres Computers werden offensichtlich bestimmte Codes anders genutzt. Das sind:

Anstelle von	geschieht folgendes:
ENQ	Zeichen weiß
BS	Blockieren der Umschaltung Klein-/Großschrift
HT	Zulassen der obigen Umschaltung
DC1	Cursor abwärts
DC2	Reverse-Modus an
DC3	Cursor in HOME-Position
DC4	INST/DEL
FS	Zeichen rot
GS	Cursor rechts
RS	Zeichen grün
US	Zeichen blau

Der auffälligste Unterschied ist der, daß beim Commodore-ASCII anstelle der Kleinbuchstaben Grafikzeichen liegen. Sollte anstelle des Normalmodus der Klein-/Großschriftmodus eingeschaltet sein, findet man anstelle der Großbuchstaben die kleinen.

Jetzt haben wir alle nötigen Kenntnisse, um die CHRGET-Routine in unserem Computer zu verstehen.

25. Die CHRGET-Routine

Das Kürzel CHRGET kommt von »Get a character«, was bei uns heißt: »Hole ein Zeichen«. Es handelt sich um eine sehr häufig benutzte Routine unseres Basic-Interpreters, die - wie schon vorhin erwähnt - komplett in der Zeropage steht. Wenn Sie mit dem SMON mal nachsehen wollen, dann geben Sie den Befehl

D 0073 008B

ein. Sie haben dann die komplette Routine vor sich:

```
0073  E6  7A    INC $7A
0075  D0  02    BNE $0079
0077  E6  7B    INC $7B
0079  AD  2502  LDA $0225
007C  C9  3A    CMP #$3A
007E  B0  0A    BCS $008A
0080  C9  20    CMP #$20
0082  F0  EF    BEQ $0073
0084  38        SEC
0085  E9  30    SBC #$30
0087  38        SEC
0088  E9  D0    SBC #$D0
008A  60        RTS
```

Eventuell sieht die Zeile 0079 bei Ihnen anders aus. Das liegt dann an den Speicherstellen $7A und $7B, welche einen Zeiger darstellen (LSB=$7A und MSB=$7B), der bei Ihnen gerade auf einen anderen Platz zeigt als auf $0225.

Diese CHRGET-Routine besteht aus drei Teilen:

Zeilen 0073 bis 0079

Weiterstellen des CHRGET-Zeigers und Einladen des dadurch angezeigten Speicherzelleninhaltes in den Akku.

Zeilen 007C bis 0082

Prüfroutinen

Zeilen 0084 bis 008A

Flaggen-Routinen

Im ersten Teil haben wir schon gleich etwas Neues vor uns: ein sich selbst veränderndes Programm. Die Speicherstelle (aus dem Basic-Eingabepuffer), aus der der Akku ein Zeichen holt, wird um 1 weitergezählt mit INC $7A. Dabei handelt es sich um das LSB der Adresse und die nächste Zeile prüft, ob ein Überlauf (255+1) stattgefunden hat:

BNE $0079.

Diese Technik kennen wir schon aus den letzten Kapiteln: Bei Überlauf wird die Z-Flagge auf 1 gesetzt und der BNE-Befehl führt keinen Sprung herbei. Den Offset von 02 können wir leicht nachrechnen: Der Programmzähler steht schon auf 0077. Die Zieladresse 0079 ist also noch 2 Byte entfernt. Hat eine Überschreitung des Höchstwertes 255 stattgefunden, dann muß das dazugehörige MSB um 1 erhöht werden. Dies tut die nächste Zeile: INC $7B

In beiden Fällen ist nun der Zeiger 7A/7B um eine Stelle weitergerückt und der Inhalt der dadurch angezeigten Speicherstelle wird in den Akku geladen. Zwei Dinge können wir uns aus diesem kurzen Programmteil merken:

1) Wie man eine 16-Bit-Zahl hoch- (oder auch herunter-) zählt und

NUL	Null	
SOH	Start of heading	Beginn des Kopfes
STX	Start of text	Textbeginn
ETX	End of text	Textende
EOT	End of transmission	Übertragungsende
ENQ	Inquiry	Anfrage
ACK	Acknowledge	Bestätigung
BEL	Bell	Klingel
BS	Backspace	Zurücksetzen
HT	Horizontal tabul.	Horizontaltabulator
LF	Line feed	Zeilenvorschub
VT	Vertical tabulator	Vertikaltabulator
FF	Form feed	Formatvorschub
CR	Carriage return	Wagenrücklauf/ Zeilenwechsel
SO	Shift out	Rückschaltung
SI	Shift in	Dauerumschaltung
DLE	Data link escape	Datenverbindungs-umschaltung
DC1-4	Device control	Gerätesteuerung
NAK	Negative acknowl.	Negativ-Bestätigung
SYN	Synchronous idle	Synchronisations-Leerlauf
ETB	End of transmission block	Ende des Übertragungsblockes
CAN	Cancel	Annullieren
EM	End of medium	Datenträgerende
SUB	Substitute	Ersetzen
ESC	Escape	Umschaltung
FS	File separator	Dateitrennzeichen
GS	Group separator	Gruppentrennzeichen
RS	Record separator	Satztrennzeichen
US	Unit separator	Einheiten-Trennz.
SP	Space	Leerzeichen
DEL	Delete	Löschzeichen

Bild 16. Die Bedeutung der Abkürzungen im ASCII-Code

2) eine Möglichkeit, Zeiger einzusetzen. Wir werden noch eine Reihe anderer Zeigertypen kennenlernen und sehen, daß es nicht immer so direkt zugeht wie hier.

Im zweiten Teil finden wir die Prüfroutinen. Die Vergleichsbefehle beschränken sich auf den Akkuinhalt, also CMP.

CMP #$3A testet, in welcher Beziehung das im Akku befindliche Zeichen zum Wert $3A = dezimal 58 steht. Erinnern wir uns an das Schema in Bild 14:

1) Commodore-ASCII-Code im Akku größer als 58, also Zeichen hinter dem Doppelpunkt (Buchstaben, Grafikzeichen, einige Sonderzeichen). Dann ist die Carry-Flagge = 1, N- und Z-Flagge sind 0.

2) Im Akku steht genau der Code 58, also der Doppelpunkt. Dann sind Carry-Bit und Z-Flagge = 1, nur die N-Flagge = 0.

3) Der Code des Zeichens im Akku ist kleiner als 58 (das wären alle Zahlen, einige Sonderzeichen und Steuerzeichen). In diesem Fall ist die N-Flagge = 1. Die beiden anderen Flaggen zeigen Null.

Der nun folgende Befehl BCS 008A überprüft die Carry-Flagge. Wenn sie gesetzt ist, wenn also der Code im Akku größer oder gleich dem eines Doppelpunktes (58) ist, springt der Programmzähler zum RTS. Der Code (und auch die Flaggen) wird unverändert zum aufrufenden Hauptprogramm weitergegeben. Zur Übung können Sie ja noch mal den Offset nachrechnen. Der Rest des Programms wird nur noch durchlaufen, wenn Codes kleiner als 58 im Akku stehen.

Die nächste Zeile CMP #$20 dient zum Vergleich des Space-Codes $20 = dezimal 32 (Leertaste). Die Flaggen treten dann, wie schon oben beim ersten Vergleich gezeigt, je nach Akku-Inhalt auf. Durch die Verzweigung BEQ $0073 erfolgt ein Rücksprung zum Beginn der CHRGET-Routine dann, wenn die Z-Flagge gesetzt ist, also ein Space-Code im Akku liegt. Somit werden die Leerzeichen einfach übersprungen und das nächste Zeichen geholt. Alle anderen Zeichen, die bis hierher durchgehalten haben, werden nun im letzten Teil der CHRGET-Routine einer Prozedur unterworfen, die ich Flaggen-Routine genannt habe.

Durch zwei aufeinanderfolgende Subtraktionen, die insgesamt den Wert im Akku unverändert lassen (es wird 256 abgezogen), wird die Carry-Flagge beeinflußt. Verfolgen wir, was da passiert:

SEC dient als Vorbereitung für die folgende Subtraktion. SBC #$30 zieht vom Akku-Inhalt $30 = dezimal 48 ab. Wir wissen inzwischen, daß das der Addition des Zweierkomplementes entspricht. Dieses ist (rechnen Sie mal nach!) 1101 0000.

Nehmen wir mal an, wir hätten den Code der Zahl 4 (also dezimal 52 oder $34) im Akku stehen. Die Rechnung sieht dann so aus:

```
52      0011  0100
        1101  0000
  +  ---------------
  (1)   0000  0100
```

Das Ergebnis ist also 4, der Übertrag wird vernachlässigt.

Als anderes Beispiel sei nun der Code für das Ausrufungszeichen im Akku (dezimal 33 = $21 = binär 0010 0001). Die Rechnung ist dann:

```
33      0010  0001
  +     1101  0000
     ---------------
        1111  0001
```

Das Ergebnis ist —15.

Alle Codes, die nicht für Zahlen stehen, haben nach dieser Subtraktion ein negatives Ergebnis im Akku hinterlassen und durch das »Borgen« das Carry-Bit gelöscht.

Nun machen wir weiter ab Zeile 0087:

SEC
SBC #$D0

Wir ziehen $D0 = dezimal 208 ab. Das Zweierkomplement ist: ...Doch da kommen wir ins Stocken! Denn dieses Zweierkomplement ist nicht mehr mit 8-Bit-Zahlen darzustellen. Schon die Zahl 208 im Binärformat (1101 0000) würde als negative Zahl angesehen werden, weil Bit 7 gleich 1 ist. Wir machen es uns einfach und sagen, daß sich das Zweierkomplement wie bisher bilden läßt, aber dabei das Carry-Bit mit einbezogen wird. Unser Zweierkomplement ist dann also: 0011 0000 und das Carry-Bit ist gelöscht. Nun nehmen wir unser erstes Beispiel. Dort war nach der Subtraktion im Akku eine 4 verblieben:

```
        0000  0100
  +     0011  0000
     ---------------
        0011  0100
```

Das ist wieder unser ursprünglicher Wert dezimal 52 = $34 = Code für die Zahl 4. Das Carry-Bit bleibt gelöscht.

Im zweiten Beispiel mit dem Ausrufungszeichen stand noch im Akku eine —15:

```
        1111  0001
  +     0011  0000
     ---------------
  (1)   0010  0001
```

Da haben wir wieder den Code für das Ausrufungszeichen ($21 = dezimal 33) im Akku und ein gesetztes Carry-Bit. Was kommt also bei der CHRGET-Routine heraus?

1) Alle Zeichen außer dem Space werden unverändert an das aufrufende Programm über den Akku weitergegeben. Space wird unterdrückt.

2) Bei allen Zeichen außer bei den Zahlen ist das Carry-Bit gesetzt.

3) Manche der aufrufenden Routinen überprüfen außer dem Zustand der Carry-Flagge auch den der Z- oder N-Flagge, die ja beim ersten CMP-Befehl ebenfalls gesetzt werden. So liefert die CHRGET-Routine noch weitere Informationen.

In der einschlägigen Literatur stoßen Sie auch auf eine Routine, die CHRGOT genannt wird. Es handelt sich dabei ebenfalls um die hier beschriebene CHRGET-Routine, nur erfolgt der Einsprung nicht bei $0073, sondern bei $0079. Der Zeiger $007A/7B wird in diesem Fall nicht weitergestellt. Das vorher schon einmal in den Akku geladene Zeichen wird damit noch einmal angesprochen (got ist die Vergangenheitsform von get).

Mit dem CHRGET-Programm haben wir eines der wichtigsten Unterprogramme unserer computerinternen Software kennengelernt. Will man sich Interpreter-Routinen zunutze machen, stolpert man ständig darüber. Außerdem aber liegt die CHRGET-Routine im RAM. Das bedeutet, daß wir sie ohne weiteres für unsere Zwecke verändern können.

26. Die indizierte Adressierung

Indizieren heißt, etwas mit einem Index (also einem Zeichen oder einer Nummer) zu versehen. Beispielsweise bezeichnet man in der Mathematik die beiden Lösungen einer quadratischen Gleichung häufig als X1 und X2. Dabei ist dann die Ziffer (1 oder 2) der Index und X ist eine indizierte Größe. Man geht also aus von einer festgelegten Grundmenge (Lösungsmenge X) und trifft durch den Index eine weitere Unterscheidung.

So ähnlich können wir uns auch die Funktion der indizierten Adressierung bei der Assembler-Programmierung vorstellen. Nehmen wir als Beispiel den Befehl

LDA $1500,X

Man spricht hier von einer absolut-X-indizierten Adressierung. Das Assemblerwort LDA ist uns bekannt: Lade den Akku. Woher soll der für den Akku bestimmte Inhalt geholt werden? Aus der Speicherzelle, die sich durch $1500 plus Inhalt des X-Registers ergibt. Steht also im X-Register zum Zeitpunkt des Befehlsaufrufes eine 5, dann wird der Akku aus Speicherzelle $1500+5, also $1505, geladen. Das X-Register kann Werte von 0 bis $FF (dez. 255) enthalten. Die Ähnlichkeit sieht also so aus:

Aus einer Gesamtmenge von 256 Adressen, die durch die Anfangsadresse (bei unserem Beispiel $1500) und die möglichen 256 Belegungen des X-Registers festgelegt sind (die Grundmenge), werden je nach X-Registerinhalt einzelne Adressen unterschieden und adressiert. Das X-Register fungiert dabei als ein Index, weswegen man auch oft die Bezeichnung »Index-Register X« in der Literatur findet.

Ebenfalls als Index-Register kann das Y-Register dienen, was zum Beispiel zum Befehl

LDX $1500,Y

führen kann. Dies ist dann eine absolut-Y-indizierte Adressierung.

Genauso wie man die normale absolute Adresse (also zum Beispiel $1500) als Basis der Indizierung durch das X- oder das Y-Register verwenden kann, ist das auch mit einer Zeropage-Adresse möglich. So gibt es zum Beispiel die Befehle

LDY $2B,X

oder

STX $19,Y

Man nennt diese Art der Adressierung dann Zeropage-absolut-X-indiziert beziehungsweise -Y-indiziert.

Weil die Zeropage aber nur 256 Adressen umfaßt, andererseits jedoch die Indexregister auch 256 Werte annehmen können, kann es geschehen (wenn man nicht aufpaßt), daß die Summe aus der Basisadresse (zum Beispiel $2B) und dem Indexregisterinhalt größer als 256 wird. Wenn zum Beispiel in dem Befehl

LDA $FE,X

der X-Registerinhalt 2 beträgt, ergäbe sich $FE+$02= $0100. In diesem Fall wird aber nicht der Inhalt von $0100 in den Akku geladen, sondern der Befehl spricht die Speicherstelle $00 an. Der Grund dafür liegt in der Tatsache, daß unser Prozessor den Befehl als 2-Byte-Befehl interpretiert - das zweite Byte ist die Zeropage-Adresse, die sich als Summe ergibt - und deswegen nur das LSB der Adresse beachtet. Von $0100 ist das LSB aber $00. Mit anderen Worten: Die Zero-page-absolut-indizierten Befehle lassen einen Zugriff nur auf die Zeropage selbst zu. Dieses Verhalten muß man beim Programmieren beachten.

Wir wollen noch mal zusammenfassen. Vier neue Adressierungsarten haben wir kennengelernt:

Absolut-X-indiziert	zum Beispiel LDA $1500,X
Absolut-Y-indiziert	zum Beispiel LDA $1500,Y
Zero-page-absolut-X-indiziert	zum Beispiel LDA $2B,X
Zero-page-absolut-Y-indiziert	zum Beispiel LDX $2B,Y

Die Verwendung des Y-Registers als Indexregister ist stark eingeschränkt. Nur bei wenigen Befehlen ist sie erlaubt (tatsächlich nur LDX und STX bei Zero-page-absolut-indizierter Adressierung). In der Tabelle 5 sehen Sie, welche bisher behandelten Befehle wie mit der indizierten Adressierung verwendet werden dürfen.

Befehl	Indizierte Adressierung absolut		Null-Seite-absolut	
	X	Y	X	Y
LDA	+	+	+	-
LDX	-	+	-	+
LDY	+	-	+	-
STA	+	+	+	-
STX	-	-	-	+
STY	-	-	+	-
RTS	/	/	/	/
INX	/	/	/	/
INY	/	/	/	/
INC	+	-	+	-
DEX	/	/	/	/
DEY	/	/	/	/
DEC	+	-	+	-
SED	/	/	/	/
CLD	/	/	/	/
BNE	/	/	/	/
ADC	+	+	+	-
CLC	/	/	/	/
SBC	+	+	+	-
SEC	/	/	/	/
BEQ	/	/	/	/
BCC	/	/	/	/
BCS	/	/	/	/
BMI	/	/	/	/
BPL	/	/	/	/
BVC	/	/	/	/
BVS	/	/	/	/
CMP	+	+	+	-
CPX	-	-	-	-
CPY	-	-	-	-
BIT	-	-	-	-
CLV	/	/	/	/
NOP	/	/	/	/
TAX	/	/	/	/
TAY	/	/	/	/
TXA	/	/	/	/
TYA	/	/	/	/
JMP	-	-	-	-
JSR	-	-	-	-

+	anwendbar
-	nicht erlaubt
/	weder absolute noch Zeropage-Adressierung möglich

Tabelle 5. Anwendbarkeit der indizierten Adressierungsarbeiten auf die bisher gelernten Assembler-Befehle

Es gibt noch zwei weitere Arten einer indizierten Adressierung, auf die wir noch zu sprechen kommen werden.

Wir wollen noch ein bißchen aufräumen: Ein paar Befehle, die bisher zu keinem Gebiet so richtig paßten, aber auch sehr wichtig sind, sollen jetzt behandelt werden.

27. Einige Nachzügler: Die Befehle BIT, CLV, NOP und TAX, TAY, TXA, TYA

BIT: Dieser Befehl heißt »Bit-Test« und paßt von daher eigentlich zu den in Kapitel 23 behandelten Vergleichsbefehlen. Die Behandlung der Flaggen ist aber völlig anders. Nehmen wir das Beispiel

BIT $1500

Folgendes passiert: Der Inhalt der Speicherstelle $1500 wird mit dem Inhalt des Akku UND-verknüpft, das Ergebnis in der Z-Flagge angezeigt und Bit 7 sowie Bit 6 von $1500 in die N- beziehungsweise die V-Flagge übertragen. Weder Akku noch Inhalt von $1500 verändern sich dabei.

Das ging ein bißchen »holterdipolter«. Sehen wir uns das jetzt mal ganz langsam Schritt für Schritt an: Zunächst die UND-Verknüpfung. Bit für Bit wird der Akku-Inhalt mit dem

Inhalt der adressierten Speicherstelle UND-verknüpft. Dabei gelten folgende Regeln:

0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 1

Nur dann also, wenn die entsprechenden Bits im Akku und in $1500 gleich 1 sind, ergibt sich bei der UND-Verknüpfung eine 1. Man stellt sowas meist in einer sogenannten Wahrheitstabelle zusammen (Tabelle 6).

UND	0	1
0	0	0
1	0	1

Tabelle 6. Wahrheitstabelle der logischen Verknüpfung UND

Nehmen wir als Beispiel mal an, im Akku stünde $0A und in der Speicherstelle $1500 wäre $09 enthalten. Die UND-Verknüpfung sieht dann so aus:

```
Akku  $0A 0000 1010
$1500 $09 0000 1001
UND ______________
          0000 1000
```

Das Ergebnis ist also $08. In der Z-Flagge wird in dem Fall, daß das Ergebnis der UND-Verknüpfung ungleich Null ist (wie hier) eine Null angezeigt, sonst eine 1.

Wir haben in unserem Zahlenbeispiel mit dem BIT-Befehl überprüft, ob die Bits 1 und 3 in Speicherstelle $1500 gelöscht sind. Dazu haben wir in den Akku eine sogenannte Maske (hier also $0A) geladen. Das Ergebnis sagt uns, daß nicht beide Bits gelöscht waren. Wäre der Inhalt von $1500 beispielsweise $10 gewesen (0001 0000), hätten wir in der Z-Flagge eine 1 gefunden. Daher der Name »Bit-Test«: Durch geeignete Maskenwahl kann praktisch jedes Bit überprüft werden. Dabei werden weder der Akku-Inhalt noch der Inhalt der angesprochenen Speicherstelle verändert.

Der BIT-Befehl hat aber noch mehr Auswirkungen: Die Bits 6 und 7 der geprüften Speicherzelle findet man nach Befehlsausführung in zwei Flaggen noch mal:

Bit 7 in der N-Flagge
Bit 6 in der V-Flagge

Damit kann man beispielsweise überprüfen, ob sich am adressierten Ort eine negative Zahl befindet. Alle drei Flaggen können ja nun mit den Branch-Befehlen abgefragt werden. Sie erkennen sicherlich schon, wie vielseitig dieser merkwürdige BIT-Befehl einsetzbar ist.

Adressierbar ist BIT entweder absolut (wie im obigen Beispiel) oder Zeropage-absolut. Je nachdem liegt er dann als 3-Byte- oder als 2-Byte-Befehl vor.

CLV: Dieser Befehl heißt »Clear oVerflow-flag«, also »lösche die Überlauf-Flagge«. Die V-Flagge war - wie Sie sich erinnern werden - unsere rote Ampel bei Rechenoperationen (siehe Kapitel 16). Es ist ein 1-Byte-Befehl mit impliziter Adressierung und interessant daran ist, daß es keinen Befehl gibt, der das Gegenteil - also das Setzen der V-Flagge - bewirkt.

NOP: NOP steht für »No OPeration«, was bedeutet »keine Tätigkeit«. Das ist der Nichtstu-Befehl. Er tut aber doch etwas: Er sorgt dafür, daß der Befehlszähler weitergezählt wird und bewirkt eine Verzögerung von zwei Taktzyklen. NOP ist ein 1-Byte-Befehl mit impliziter Adressierung. Er wird in fertigen Programmen nur selten verwendet: Zur Erzeugung einer kurzen definierten Verzögerung. Meist gebraucht man ihn bei der Erstellung eines Programmes als Platzhalter oder bei der Fehlersuche, um zum Beispiel unerwünschte Sprünge zu ersetzen.

Die Transporteure: TAX, TAY, TXA und TYA

Ab und zu ist es nötig, Registerinhalte untereinander auszutauschen. Viele Dinge (Addition, Subtraktion und so weiter) können nur im Akku geschehen. Wenn wir eine solche Operation beispielsweise mit dem Inhalt des X-Registers ausführen wollen, verschieben wir diesen Inhalt mit dem Befehl TXA. »Transfer X into Accumulator« also »übertrage X-Register in den Akku« bedeutet das. Analog verwendet man TYA, um Y-Register-Inhalte in den Akku zu schieben oder für den umgekehrten Weg TAY beziehungsweise TAX (Akkuinhalt ins Y- beziehungsweise ins X-Register schieben). Genau genommen wird nicht übertragen, sondern nur kopiert: Die Register, aus denen verschoben wird, bleiben unverändert. Weil die jeweiligen Zielorte der Verschiebung (Akku, X- oder Y-Register) vom neuen Inhalt überschrieben werden, können sich auch Flaggen ändern. Betroffen sind von dieser Möglichkeit die N- und die Z-Flagge. Alle vier Befehle bestehen aus einem Byte und können natürlich nur implizit adressiert werden.

28. So springen die Assembler-Alchimisten: JMP, JSR

JMP und JSR entsprechen ungefähr den vom Basic her bekannten Befehlen GOTO und GOSUB.

JMP kommt von »JuMP to address«, also »springe zur angegebenen Adresse«. Nehmen wir uns wieder ein Beispiel vor:

```
JMP    $1500
```

bewirkt einen Sprung zur Adresse $1500. Das funktioniert so: In den Programmzähler werden LSB und MSB der Zieladresse geladen. Das war dann auch schon der Sprung, denn der Programmzähler ist der Pfadfinder des Computers: Die Adresse, die dort steht, wird als nächste bearbeitet. Schalten Sie doch mal den SMON ein (oder einen anderen Monitor) und sehen Sie sich das mit folgenden Befehlen an:

```
1400   JMP   $1500
```

Dort unterbrechen wir den Computer mit

```
1500   BRK
```

So weit, so gut: Wir starten mit dem SMON-Kommando G 1400 und erhalten eine Registeranzeige mit dem Programmzählerstand 1501. Genau das hatten wir ja erwartet. Weniger durchschaubar ist das folgende Beispiel:

```
1400   LDA   #$00
1402   LDX   #$16
1404   STA   $1300
1407   STX   $1301
140A   JMP   $(1300)
```

Dazu gehört dann noch die Programmzeile:

```
1600   BRK
```

Wenn Sie das genauso eingegeben haben und dann mittels G 1400 starten, erhalten Sie eine Registeranzeige mit dem Programmzählerstand 1601.

Schon an der neuen Schreibweise des Argumentes in Zeile 140A werden Sie bemerkt haben, daß hier nicht mehr die normale absolute Adressierung wie zuvor angewendet wird. Dies ist eine neue Form: Die **indirekte Adressierung**. Indirekt deswegen, weil wir nicht mehr direkt die Zieladresse angeben, sondern einen sogenannten Vektor. Ein Vektor besteht aus zwei aufeinander folgenden Speicherzellen (hier also $1300 und $1301), die in der Form LSB/MSB die eigentliche Zieladresse enthalten. Das LSB von $1600 ist $00. Das haben wir über den Akku nach $1300 geladen.

Das MSB $16 kam durch das X-Register an seinen Platz $1301:

```
         Zieladresse 16     00
                     MSB    LSB
                     ↑      ↑
         Vektor      1301   1300
```

Das ist die Methode der toten Briefkästen, die in Kreisen der Assembler-Alchimisten anscheinend genauso beliebt ist wie bei Agenten. So wie diese im hohlen Baum die Treffpunktanschrift hinterlegt finden, verläßt sich unser Computer auf die Speicherstellen $1300 und $1301 für die Angabe der Zieladresse.

Diese Art der Adressierung ist im wahrsten Sinn des Wortes ein Unikum: Es gibt sie nämlich nur für den JMP-Befehl! Davon wird allerdings dann auch recht häufig Gebrauch gemacht, zum Beispiel im Betriebssystem unseres Computers. Aber darüber und über die Vektoren, die dazu verwendet werden, soll ein andermal berichtet werden.

Wir dürfen nämlich nicht den anderen Sprungbefehl JSR vergessen. JSR steht für »Jump to SubRoutine«, was eingedeutscht etwa bedeutet »springe zum Unterprogramm«. Genauso wie in Basic Unterprogramme durch GOSUB Zeilennummer aufgerufen werden, kann das auch hier geschehen durch JSR Adresse. Hier ist nur die absolute Adressierung möglich. Das erste Beispiel soll uns zeigen, wie dieser Befehl funktioniert:

```
1400   JSR    $1500
```

Dort soll dann erstmal stehen:

```
1500   BRK
```

Noch nicht starten!! Zunächst einmal verzeihen Sie mir diese Programmierer-Todsünde: Aus einem Unterprogramm heraus den Programmablauf zu beenden! Ich werd's auch nie wieder tun. Hier geschieht das nur zu Lehrzwecken. Was läuft ab: Der Programmzählerinhalt plus 2 wird auf den Stapel gelegt und dann die Adressse $1500 in den Programmzähler geladen. Ebenso kurz wie unklar! Was ist denn ein Stapel? Also langsam, Schritt für Schritt.

Der Sinn von Unterprogrammen ist ja, daß der Computer nach Ende der Bearbeitung wieder ins aufrufende Hauptprogramm zurückkehrt. Er muß sich aber dazu irgendwo merken, von wo aus er zum Unterprogramm gesprungen ist. Dazu verwendet er den Stapel. Das ist ein Speicherbereich ($0100 bis $01FF), der direkt vom Prozessor aus verwaltet wird. Die genaue Architektur und Handhabung dieses »Prozessor-Stack« werden wir noch in einer späteren Folge kennenlernen. Uns soll hier nur interessieren, daß es einen Zeiger gibt, der auf den nächsten freien Platz im Stapel weist und daß dieser Speicher von oben nach unten gefüllt wird (wie in Basic bei den Strings). Wenn Sie mit Hilfe des SMON mal in den Stapel hineinsehen wollen, dann geben Sie doch mal ein M 0100 01FF. Was nun genau bei Ihnen drin steht, ist sehr von der vorherigen Nutzung Ihres Computers abhängig. Der Mikroprozessor nutzt den Stapel bei sehr vielen Tätigkeiten. Es kommt auch nur auf den Teil des Stapels an, der durch den Stapelzeiger als gefüllt bezeichnet wird. Der Stapelzeiger wird beim SMON in der Registeranzeige als SP angezeigt. Wenn Ihr Stapelzeiger (prüfen Sie das doch mal durch Eingabe von R) nun zum Beispiel F6 zeigt, dann bedeutet das, daß alle Stapelplätze von $01F6 an abwärts frei und die oberhalb bis $01FF besetzt sind. Beim Nachsehen mit M 01F0 01FF finden Sie dann beispielsweise:

```
:01F0  20  00  20  AA  C1  FA  C0  46
                               --
:01F8  E1  E9  A7  A7  79  A6  9C  E3
```

Die Speicherstelle, auf die der Stapelzeiger weist, ist unterstrichen. Nun starten wir mit G 1400 unser kleines verbotenes Testprogramm. Es meldet sich die Registeranzeige.

Im Stapelzeiger steht jetzt F4 (oder eben Ihr vorhergegangener SP minus 2). Wenn wir nun wieder im Stapel nachsehen mit M 01FO 01FF, dann finden wir im Gegensatz zur obigen Anzeige nun:

```
:01F0  20  AA  C1  FA  C0  02  14  46
                       --  ↑↑  ↑↑
:01F8  E1  E9  A7  A7  79  A6  9C  E3
```

Unterstrichen ist wieder das Ziel des Stapelzeigers, der jetzt zwei Plätze weitergerückt ist, um der durch Pfeile gekennzeichneten Adresse 1402 (als LSB/MSB) Raum zu schaffen. $1402 ist das letzte Byte des JSR-Befehls. Wie wir den Programmzähler kennen, ist er im allgemeinen immer einen Schritt voraus. Hier liegt er aber einen zurück, falls er nach Beendigung des Unterprogrammes an der notierten Adresse weitermacht. Dazu kommen wir gleich noch. Was wir am Programmzähler aber auch noch nach Ablauf unseres kurzen Beispielprogrammes ablesen können, ist die Tatsache, daß die Sprungadresse $1500 in ihn geschrieben wird, somit der Sprung dann also stattgefunden hat.

Nun bauen wir das kleine Programm etwas um:

```
1400   JSR    $1500
1403   BRK
```

Das Unterprogramm soll nur aus dem Rücksprung bestehen:

```
1500   RTS
```

Verlangen Sie nun noch vor dem Start eine Registeranzeige mit R und merken Sie sich den Wert des Stapelzeigers. Dann starten Sie das Programm mit G 1400 und achten Sie auf die neue Registeranzeige. Zwei Dinge interessieren uns:

1) Der Wert des Stapelzeigers ist unverändert geblieben.
2) Der Programmzähler weist nun auf $1404.

Wenn Sie nun noch mal mit dem M-Befehl des SMON in den Stapel sehen, werden Sie unter Umständen zwar noch die Adresse $1402 dort finden (dann nämlich, wenn wir den Stapel seit dem letzten Programm nicht verändert haben). Wie Sie aber inzwischen wissen, hätte durch den neuen JSR-Befehl noch mal 1402 dort eingetragen sein müssen. Das stand da auch einige Mikrosekunden lang... bis der RTS-Befehl wirksam wurde. RTS macht ziemlich viel:

1) RTS holt die auf dem Stapel gespeicherte Adresse ab, und schreibt sie in den Programmzähler.
2) RTS vermindert dabei den Stapelzeiger um 2.
3) RTS addiert zum Programmzähler eine 1.

Deswegen kann das Programm also bei $1403 weiterlaufen und der Programmzähler nun hinter dem BRK-Befehl stehen.

Machen Sie doch mal etwas anscheinend total Verrücktes: Starten Sie mit G 1500. Es gibt da zwei Möglichkeiten, was geschehen kann: Entweder stand da noch vom ersten unterbrochenen Testprogramm die Adresse 1402. Dann endete nun alles mit einer Registeranzeige, bei der der Stapelzeiger um 2 höher gerutscht ist.

Oder da stand diese Adresse nicht mehr. Dann befinden Sie sich nun wieder im Basic. Wieso eigentlich? Als nächste Adresse finden Sie auf dem Stapel $E146 (dez.57670). Diese Adresse + 1 wird ja durch RTS in den Programmzähler gerufen. Ein Sprung an diese Adresse ist ein Sprung in ein Programm des Betriebssystems. Haben Sie ein ROM-Listing? Dann sehen Sie mal nach: Dort steht der Befehl...RTS. Dies neuerliche RTS holt nun jedenfalls die nächste Adresse vom Stapel: $A7E9 (dez.42985). Diese Adresse + 1 im Programmzähler führt unseren Computer in die Basic-Interpreter-Schleife, also ins Basic zurück.

Wir haben so viel über den Stapel gehört, daß wir JSR fast schon wieder aus den Augen verloren haben. Deswegen noch mal eine kurze Übersicht:

a) JSR speichert den Programmzählerwert des letzten Bytes des Befehls auf dem Stapel zum Beispiel $1402,

b) stellt dabei den Stapelzähler um 2 zurück zum Beispiel von $F6 nach $F4
c) schreibt in den Programmzähler die angegebene Zieladresse, zum Beispiel $1500
d) Das Unterprogramm wird abgearbeitet bis der RTS-Befehl auftaucht.
e) Dann wird die gemerkte Adresse +1 in den Programmzähler geschrieben, zum Beispiel $1402+1=$1403
f) und dabei der Stapelzähler wieder um 2 erhöht, zum Beispiel von $F4 wieder zu $F6
g) Das Programm läuft nun wieder nach dem JSR-Befehl weiter, zum Beispiel also bei $1403.

Nun sollte eigentlich auch klar sein, warum ein Aussprung aus einem Unterprogramm oder ein Abbruch im Unterprogramm eine Programmierer-Todsünde ist: Der Stapelzeiger wird nicht zurückgestellt. Die gemerkte Rücksprungadresse versauert allmählich auf dem Stapel. Noch schlimmer sind solche Sachen in einer Schleife, wo mehrfach aus dem Unterprogramm ausgebrochen wird: Hier ist der Stapel bald voll Müll und der Computer beendet seine Zusammenarbeit mit dem Programmierer. Weil aber Basic-Programme nichts anderes sind als eine Folge von Maschinenprogrammen, die je nach Befehl durch den Interpreter aneinandergereiht werden, ist das auch in Basic eine Todsünde. Wir wollen aber nicht so hart mit uns umgehen: Wenn wir gelernt haben, wie man mit speziellen Assembler-Befehlen im Stapel herumschaufeln kann, dann haben wir bei richtiger Anwendung von vorneherein jedenfalls in diesem Punkt die Absolution erhalten.

29. Alles fließt: Fließkommazahlen

Jeder, der tiefer in die Geheimnisse der Assembler-Alchimie eindringen will, muß sich vertraut machen mit der häufigsten Art der Zahlenverarbeitung in unserem Computer. Das ist die Handhabung von Fließkommazahlen (auch Gleitkommazahlen genannt). Wir werden dazu folgende Fragen zu klären haben:
1) Was sind Fließkommazahlen?
2) Wie sehen sie im binären Zahlensystem aus?
3) Wie behandelt unser Computer positive und negative Fließkommazahlen?
4) Wie können wir als Programmierer Einfluß nehmen auf die Verarbeitung dieser Zahlen im Computer?

Die Behandlung dieser vier Fragen wird uns eine ganze Weile beschäftigen. Fangen wir mit der ersten an: In Standardwerken der Mathematik werden Sie lange suchen müssen, um den Begriff »Fließkommazahl« zu finden. Im deutschen Sprachraum gibt es häufiger die Bezeichnung »wissenschaftliche Zahlendarstellung«. Das klingt sehr hochgestochen und ist eigentlich ganz einfach; die Zahl 1000 kann man auf verschiedene Weise darstellen:

$1000 = 10 * 10 * 10 = 10^3$ (in Basic 10↑3)

Die hochgestellte Zahl (in Computerschreibweise: Die Zahl hinter dem Hochpfeil) ist hier gleich der Anzahl der Stellen minus 1 (1000 hat vier Stellen, also ist die Hochzahl eine 3). Diese Hochzahl nennt man Exponent (vom lateinischen exponere = anzeigen, herausheben). Nehmen wir nun einige andere Zahlen:

200 = 2 * 100 = 2 * 10↑2

oder

2500 = 2,5 * 1000 = 2,5 * 10↑3

Ich glaube, jetzt beginnt es Ihnen klarzuwerden, daß man auf diese Art wohl alle Zahlen irgendwie darstellen kann. Man dröselt die Zahlen auseinander, bildet ein Produkt, von dem der eine Multiplikator durch 10 teilbar ist (durch die Basis unseres normalen Zahlensystems). Genauer gesagt: Ein Faktor (also in den Beispielen 1000 oder 100) ist darstellbar als Potenz von 10. Der andere Faktor (in den Beispielen 1 oder 2 oder 2,5) wird Mantisse (vom lateinischen manitissa = Zugabe, Anhang, Schleppe) genannt. Sehen wir uns noch mal 2500 an:

```
2500 =    2,5 * 1000 =    2,5 * 10↑3
     =     25 *  100 =     25 * 10↑2
     =    250 *   10 =    250 * 10↑1
     =   2500 *    1 =   2500 * 10↑0
```

Das letzte war nur der Vollständigkeit halber, denn irgendeine Zahl hoch 0 ist immer 1. Man kann auch aus der 2500 folgendes machen:

```
2500 = 0,25  *  10000 = 0,25  * 10↑4
oder = 0,025 * 100000 = 0,025 * 10↑5
```

und so weiter. Oder anders herum:

```
2500 =  25000 * 0,1  =  25000 * 10↑-1
     = 250000 * 0,01 = 250000 * 10↑-2
```

und so weiter.

Dabei bedeutet:

$10^{-2} = 1/10^2 = 0,01$

Man kann sich das merken, indem man die Anzahl der Stellen zählt, um die man das Komma verschiebt. Diese Anzahl addiert man dann zur Hochzahl. Zur Erläuterung:

$0,12345 = 1,2345 * 10^{-1}$

Wir haben das Komma um eine Stelle nach rechts gerückt, weshalb wir die Hochzahl -1 schreiben müssen (vorher war da nämlich unsichtbar die Hochzahl 0: und 10↑0=1).

$0,12345 = 123,45 * 10^{-3}$

Hier wurde das Komma um drei Stellen nach rechts verschoben. Daher der Exponent -3. Sie sehen folgenden Zusammenhang:

Komma eine Stelle nach rechts verschoben: Exponent + (-1).
Zum Beispiel
$0,1234*10^{-2} = 1,234*10^{-3}$

Komma eine Stelle nach links verschoben: Exponent +1.
Zum Beispiel
$3,14*10^{-2} = 0,314*10^{-1}$

Verstehen Sie nun, warum man diese Art der Zahlendarstellung Fließkomma- oder Gleitkommazahlen nennt?

Vielleicht sehen Sie aber noch nicht den Sinn der Fließkommazahlen ein. Dazu gebe ich Ihnen zwei einsichtige Beispiele. Der Atomkern eines Heliumatoms wiegt etwa (halten Sie sich fest):
0,000 000 000 000 000 000 000 000 006 643 kg.

Sehr unbequem, diese ganzen Nullen immer mitzuschleppen. Wir verschieben deshalb das Komma um 27 Stellen nach rechts und schreiben dann

$6,643*10^{-27}$ kg.

2. Beispiel: Wir haben einen Ballon mit Helium gefüllt. Bei normalen Temperatur- und Luftdruckbedingungen befinden sich in einem Kubikzentimeter im Ballon ungefähr (nochmal festhalten!):
26 900 000 000 000 000 000 Heliumatome

Wieder eine recht unangenehme Nullschlepperei. Wir verschieben das Komma um 19 Stellen nach links und erhalten $2,69*10^{19}$ Heliumatome. Fein, nicht wahr?

Abgesehen von der höheren Bequemlichkeit: Der Computer müßte allerhand Speicherplatz zur Handhabung der vielen Nullen bereitstellen. Mit BCD-Zahlen könnten wir zwar jede Zahl erfassen, hätten aber immer unterschiedlich viele Bytes zu verarbeiten. Wenn wir Fließkommazahlen verwenden, können wir – wie Sie noch sehen werden – jede (na sagen wir mal: fast jede) Zahl in der gleichen Anzahl Bytes aufbewahren.

Vom Basic her kennen Sie Fließkommazahlen auch (hier wird das Komma allerdings durch den Punkt ersetzt, entsprechend der angloamerikanischen Schreibweise). Das sind die, wo man zum Beispiel schreibt 6.02E23 oder 6.02E+23, was dann bedeutet $6{,}02*10^{23}$. »E« steht dort für Zehnerexponent. Durch die Art, wie Fließkommazahlen im normalen Computerdasein gespeichert werden, ergeben sich obere und untere Grenzen. Die höchste in Basic verarbeitbare Zahl im C 64 ist

$+1.70141183*10^{38}$

Größere Zahlen verursachen in Basic einen OVERFLOW ERROR. Was in Maschinensprache mit größeren Zahlen geschieht, ist weitgehend unsere Sache. Die dem Betrag nach kleinste verarbeitbare Zahl ist

$\pm 2.93873588*10^{-39}$

In Basic arbeitet bei Unterschreitung der Computer einfach mit einer Null weiter. Für die Behandlung in Maschinensprache sind ebenfalls wir als Programmierer verantwortlich.

Für diesmal sei's genug der Zahlenspiele: Später werden wir uns weiter mit Fließkommazahlen befassen.

30. Die USR-Funktion

Wieder einmal soll uns das Zusammenspiel von Basic und Maschinensprache beschäftigen. Einen Aufruf von Maschinenroutinen - nämlich den mit SYS - haben wir schon kennengelernt. Wir POKEten die zu übergebenden Werte an die Abrufspeicherstellen. Bei diesen Werten hat es sich um einfache Integerzahlen gehandelt, zum Beispiel die Anzahl der Glieder einer zu summierenden arithmetischen Reihe. Was tun wir aber, wenn wir Fließkommavariable an ein Maschinenprogramm übermitteln wollen? Gewiß, werden Sie sagen, lernen wir das ja noch und können dann entsprechende POKE-Kommandos geben. Damit haben Sie auch recht, nur ist das dann der »harte« Weg. Es gibt auch einen problemlosen »weichen« Weg, nämlich das USR-Kommando.

USR ist ein Basic-Befehl und rührt her von »USeR callable machine language subroutine«, also »durch den Benutzer aufrufbares Maschinensprachunterprogramm«. Darin liegt eigentlich noch nichts Neues gegenüber dem SYS-Befehl. Im Gegensatz zu SYS - wo das Argument die Einsprungadresse des Maschinenprogrammes ist - übergibt USR als Argument eine beliebige Fließkommavariable in festgelegter Form an eine sehr nützliche Speicherstellenkombination, den Fließkomma-Akkumulator 1, von uns künftig einfach FAC genannt. Der FAC belegt die Speicherstellen 97 bis 102 ($61 bis $66). Wenn das eventuell in Basic benötigte Ergebnis dort auch in der vorgeschriebenen Form abgelegt wird, kann es im Basic-Programm weiterverwendet werden. Keine Angst, dazu kommen wir bei der weiteren Behandlung der Fließkommazahlen noch ganz ausführlich zu sprechen. Jetzt soll uns das noch nicht belasten. Als Argument kann man nämlich auch irgendeine bedeutungslose Größe, ein sogenanntes Dummy angeben, das dann gar nicht weiter verwendet wird. Der USR-Befehl dient in diesem Fall lediglich dem bequemen Ansteuern eines Maschinenprogrammes.

Woher weiß unser Computer beim USR-Befehl, welche Maschinenroutine er im 64-KByte-Speicher bearbeiten soll? Beim SYS-Befehl ist das klar: Das Argument sagt es:

SYS 24345

läßt den Programmzähler auf dez.24345 zeigen. Aber wenn wir eingeben:

USR(24345)

dann packt der Computer die Zahl 24345 als Fließkommavariable in den FAC und meldet dann einen SYNTAX ERROR. Das liegt daran, daß der Basic-Interpreter beim USR-Befehl einen der oben kennengelernten indirekten Sprünge vollführt:

JMP ($311)

$311/312 (in dezimal 785/786) ist also ein Vektor, und der weist im Normalfall zu einer Routine, die den SYNTAX ERROR ausgibt (dez. 45640). Bevor wir also den USR-Befehl geben, müssen wir in diesen Vektor die Startadresse unserer Maschinenroutine schreiben:

```
dez. 24345 = $5F19
LSB $19 = dez. 25 in Speicher 785 mit
           POKE 785,25
MSB $5F = dez. 95 in Speicher 786 mit
           POKE 786,95
```

Jetzt weiß der Computer, wohin er beim USR-Aufruf springen soll, und solange, bis wir den Vektor wieder ändern, führt er bei jedem USR-Befehl unser bei 24345 stehendes Maschinenprogramm aus. Wir müssen nur noch dafür sorgen, daß dort dann auch wirklich eines anfängt. Ein Beispiel werden wir nachher noch behandeln.

Die Struktur des C 64-Speichers ist vereinfacht schon zu Beginn dieses Kurses gezeigt worden. Dabei tauchten zwei ROM-Bereiche auf, die wir Basic-Interpreter und Betriebssystem genannt haben. Diese Unterteilung ist nicht ganz korrekt. Wenn Sie über ein ROM-Listing verfügen und beispielsweise das Ende des ROM-Bereiches von $A000 bis $BFFF sowie den Anfang des oberen ROM ($E000 bis $FFFF) untersuchen, dann stellen Sie fest, daß ab dez. 49087 ($BFBF) die Basic-Funktion EXP bearbeitet wird. Der letzte Befehl vor $C000 beendet diese Funktion aber nicht etwa, sondern dort steht:

JMP $E000

Tatsächlich läuft ab $E000 bis $E042 die Bearbeitung der EXP-Funktion munter weiter, und auch danach finden sich allerlei Basic-Befehle (SIN, COS und so weiter). Da liegt also keine klare Trennung vor, sondern ein Mischmasch. Wir sollten uns vielleicht angewöhnen - statt vom Interpreter und dem Betriebssystem -, vom unteren und oberen ROM-Bereich zu sprechen.

31. Der harte Kern: Noch mal Speicherfragen

Eine andere Unterscheidung ist dagegen sinnvoll: Wie einige Besitzer neuerer Commodore 64 sicherlich bemerkt haben, sind Teile der ROM-Routinen im Laufe der Zeit verändert worden. Hauptsächlich geht es bei den aktuellen Neuerungen dieser internen Maschinenprogramme um die Farbgebung der Zeichen. Man kann eigentlich nie so recht wissen, was den Software-Planern von Commodore noch alles einfällt. Jedenfalls können deren Ideen manchmal recht dramatische Folgen haben, nämlich dann, wenn Sie ein fabelhaftes Maschinenprogramm gebaut haben, welches ROM-Routinen direkt verwendet. Der Programmierer spielt auf diese Weise eine milde Form des russischen Roulettes. Glücklicherweise halten sich die Änderungen in Grenzen, und wir dokumentieren unsere Programme ja auch immer gut (Sie etwa nicht??). Notwendige Umbauten können also leicht vonstatten gehen.

Ganz ohne ROM-Routinen-Verwendung kommt man eigentlich kaum aus. Es gibt aber einen ROM-Bereich, für den Commodore verspricht, keinerlei Änderungen durchzuführen: die Kernel-Sprungtabelle.

Das ist ein Programmbereich ($FF81 bis $FFF5), in dem 39 JMP-Befehle enthalten sind (zum Teil in absoluter, aber auch in indirekter Adressierung). Jeder dieser Sprungbe-

fehle weist auf die Einsprungadresse eines Maschinenprogrammes. Da finden sich alle wichtigen Ein/Ausgabe-Operationen, Systemtakt- und Uhrsteuerungen und anderes mehr. Wir werden uns nach und nach damit vertraut machen. In der Tabelle 7 sind die Kernel-Adressen und ihre Funktion aufgeführt. Manche davon können ohne jede Vorbereitung benutzt werden, andere brauchen bestimmte Routinen oder Angaben, um sinnvoll zu arbeiten.

Die Absicht von Commodore ist es, daß jeder Aufruf von zum Beispiel $FFD2 die Ausgabe eines Zeichens bewirkt, und zwar unabhängig davon, welchen Computer in welcher Version wir benutzen. Das Programm, welches diese Zeichenausgabe letztendlich ausführt, kann sich ändern, kann in ganz andere Speicherbereiche gelegt werden. An der Stelle $FFD2 wird aber immer ein JMP mit der Einsprungadresse stehen. Leider ist diese Sprungtabelle viel zu knapp gehalten. Es gibt so viele interessante ROM-Routinen, die wir alle ohne dieses Sicherheitsnetz anspringen müssen.

Adresse HEX	Adresse dezimal	Name	Funktion
FF81	65409	CINT	Prüfen der TV-Norm, Berechnung der Taktfrequenz
FF84	65412	IOINIT	Ein/Ausgabe-Reset
FF87	65415	RAMTAS	Prüfen auf freien Basic-RAM
FF8A	65418	RESTOR	Initialisieren der I/O-Vektoren
FF8D	65412	VECTOR	Lesen und Setzen der I/O-Vektoren
FF90	65424	SETMSG	Setzen des Ausgabe-Modus
FF93	65427	SECOND	Ausgeben der Sekundäradresse nach LISTEN
FF96	65430	TKSA	Ausgabe der Sekundäradresse nach TALK
FF99	65433	MEMTOP	Lesen/Setzen des Speicherendes
FF9C	65436	MEMBOT	Lesen/Setzen des Speicheranfangs
FF9F	65439	SCNKEY	Abfragen der Tastatur
FFA2	65442	SETTMO	Setzen der Time-Out-Flagge
FFA5	65445	ACPTR	Zeichen vom seriellen Port in Akku lesen
FFA8	65448	CIOUT	Zeichen vom Akku auf seriellen Port ausgeben
FFAB	65451	UNTLK	Sendet UNTALK an seriellen Bus
FFAE	65454	UNLSN	Sendet UNLISTEN an seriellen Bus
FFB1	65457	LISTEN	Sendet LISTEN an Geräte per seriellen Bus
FFB4	65460	TALK	Sendet TALK an Geräte per seriellen Bus
FFB7	65463	READST	Liest I/O-Status in den Akku
FFBA	65466	SETLFS	Festlegung der Parameter für OPEN
FFBD	65469	SETNAM	Festlegung des Filenamens
FFC0	65472	OPEN	Öffnet spezifizierten File
FFC3	65475	CLOSE	Schließt spezifizierten File
FFC6	65478	CHKIN	Öffnet einen Eingabekanal
FFC9	65481	CHKOUT	Öffnet einen Ausgabekanal
FFCC	65484	CLRCHN	Schließt Ein- und Ausgabekanäle
FFCF	65487	CHRIN	Holt vom aktiven Eingabekanal ein Zeichen in den Akku
FFD2	65490	CHROUT	Sendet Akku-Inhalt auf aktiven Ausgabekanal
FFD5	65493	LOAD	LOAD und VERIFY von Programmen
FFD8	65496	SAVE	Speichern von Programmen
FFDB	65499	SETTIM	Uhrzeit setzen
FFDE	65502	RDTIM	Uhrzeit lesen
FFE1	65505	STOP	STOP-Taste abfragen
FFE4	65508	GETIN	Zeichen aus dem Tastaturpuffer in den Akku lesen
FFE7	65511	CLALL	Schließen aller Kanäle und Files
FFEA	65514	UDTIM	Uhr um 1/60 Sekunde weiterzählen
FFED	65517	SCREEN	Lesen des Bildschirmformates
FFF0	65520	PLOT	Lesen/Setzen der Cursor-Position
FFF3	65523	IOBASE	Lesen der Startadresse der Ein- und Ausgabebausteine

Tabelle 7. Die Sprungtabelle für Kernel-Routinen

32. Die Urzelle eines Programmprojektes

Wir sind jetzt soweit, daß wir die Urzelle eines Programmprojektes, welches uns eine lange Zeit begleiten wird, aufbauen können. Wir wollen etwas unter den Teppich kehren. Der Teppich, das sind die uns bislang nicht zugängigen RAM-Bereiche unter den ROMs. Haben Sie das nicht auch schon mal erlebt, daß Sie während einer Programmarbeit plötzlich feststellen, Sie benötigen zum Beispiel für eine Zwischenrechnung ein weiteres Programm, oder Sie wälzen Listen und denken sich, ein kleiner Hilfsbildschirm wäre jetzt von Nutzen, oder....

Mit diesem heute zu startenden Programm wäre all das und noch viel mehr realisierbar. Es soll auf einfache Weise beliebige Speicherbereiche unters ROM schieben und sie wieder hervorholen können.

Natürlich braucht die Entwicklung dieses Projektes einige Zeit, zumal wir noch vieles lernen müssen. Deswegen sind wir in dieser ersten Urzelle noch sehr eingeschränkt: Wir verschieben zuerst einmal nur eine Bildschirm-Kopfzeile unter den oberen ROM-Bereich. Auch in dieser einfachsten Version gibt es noch einige Programmteile, die Sie erst nach und nach verstehen werden. Aber irgendwann müssen wir ja mal anfangen, Nägel mit Köpfen zu machen.

Unser Maschinenprogramm soll durch die USR-Funktion aufgerufen werden. Wie wir es gelernt haben, muß deshalb vor dem ersten Aufruf eine Initialisierung durch Belegen des USR-Vektors mit unserer Startadresse stattfinden. Die Startadresse soll $02B6 (dez. 694) sein, denn dort gibt es einen freien RAM-Bereich bis inklusive $02FF (dez. 767), der weder andere Programme noch Kassettenoperationen stört. Das MSB $02 ist dezimal auch 2 und wird nach 786 gePOKEt:

POKE 786,2

Das LSB $B6 ist dezimal 182 und soll in 785 geschrieben werden:

POKE 785,182

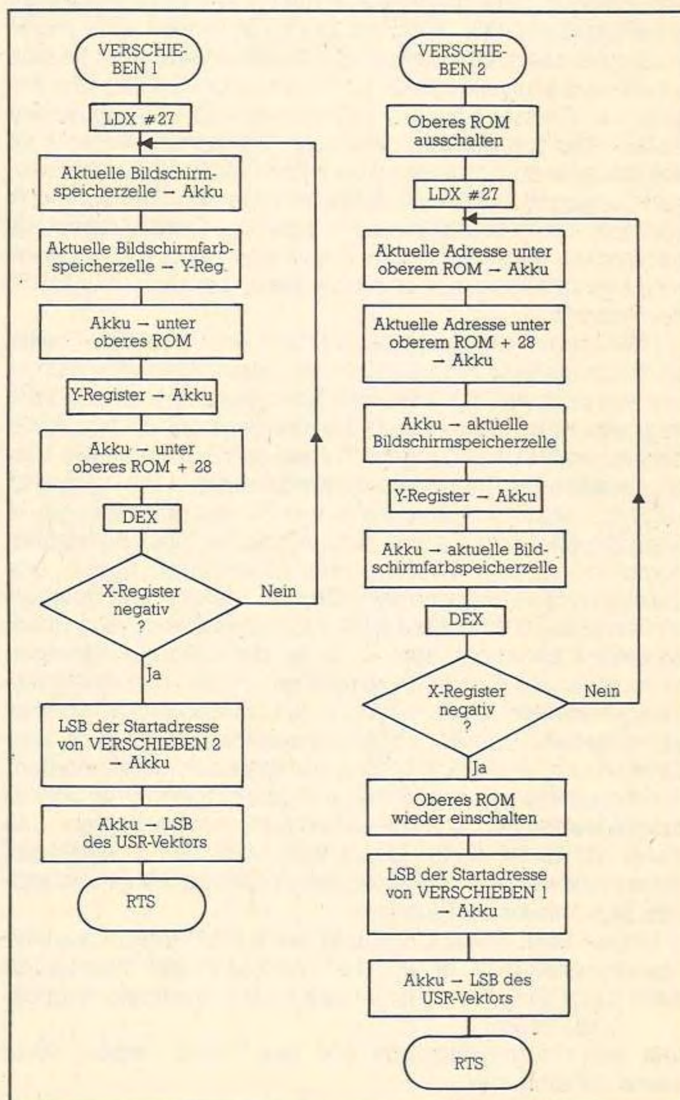


Bild 17. Das Flußdiagramm zu dem im Text erklärten Programm

Damit ist der USR-Vektor gestellt und wir brauchen uns nicht mehr weiter darum zu kümmern: Jeder USR-Aufruf wird nun den Start des Programmes bewirken. Nun zum Programm selbst. In Bild 17 finden Sie ein Flußdiagramm dazu.

Zunächst konstruieren wir den Teil, der die erste Bildschirmzeile nach $E000 und folgende Speicherstellen schiebt. Das X-Register verwenden wir als Index und laden es mit dez.39 = $27.

Schalten Sie also den SMON ein und starten Sie den Assembler mit:

```
A 02B6
```

Dann geben Sie ein:

```
02B6   LDX   #$27
```

Nun packen wir das letzte Zeichen der obersten Bildschirmzeile in den Akku:

```
02B8   LDA   $0400,X
```

In das Y-Register legen wir die dazugehörige Farbe aus dem Bildschirmfarbspeicher:

```
02BB   LDY   $D800,X
```

Den Akkuinhalt - also die Bildschirminformation - legen wir nach $E000+$27:

```
02BE   STA   $E000,X
```

Dasselbe tun wir mit dem Farbcode, der ab $E028+$27 abwärts gespeichert wird. Leider kann man STY nicht X-indiziert-absolut adressieren (siehe Tabelle 5). Deshalb schieben wir zuerst den Y-Registerinhalt in den Akku:

```
02C1   TYA
02C2   STA   $E028,X
```

Damit ist das letzte Zeichen der Kopfzeile verschoben. Wir zählen das X-Register um 1 herunter:

```
02C5   DEX
```

Der X-Index weist nun auf das vorletzte Zeichen, mit dem sich alles ab $02B8 wiederholt. Wenn das X-Register bis 0 heruntergezählt ist, weist es auf das erste Zeichen der Kopfzeile. Die Schleife muß dann noch einmal durchlaufen werden und ein weiteres Herabzählen des X-Registers erzeugt $FF, was zum Setzen der N-Flagge führt. Das ist dann unser Signal, daß die gesamte Kopfzeile übertragen wurde. Die N-Flagge wird durch den BPL-Befehl getestet:

```
02C6   BPL   $02B8
```

So weit, so gut. Wir hätten natürlich auch das X-Register von 0 an hochzählen können. Zum Beenden der Schleife wäre dann aber ein CPX-Befehl erforderlich gewesen, der jedesmal den X-Registerinhalt mit der Zahl $27 vergleicht.

MERKE: Indexregister in Schleifen abwärts zu zählen, kann Rechenzeit einsparen!

Ab $02CE soll der umgekehrte Vorgang, also das Zurückschieben der vorher gespeicherten Kopfzeile in den Bildschirmspeicher geschehen. Das einfachste wäre es sicherlich, diesen Programmteil mit einem weiteren USR-Kommando zu starten. Das sähe dann so aus:

1.USR-Befehl - schiebt Kopfzeile unter oberes ROM
2.USR-Befehl - holt Kopfzeile zurück in Bildschirmspeicher
3.USR-Befehl - schiebt wieder Kopfzeile unter ROM
4.USR-Befehl - holt sie wieder zurück und so weiter.

Weil aber das Umstellen des USR-Vektors durch POKEs vom Basic aus lästig ist, tun wir das einfach immer am Ende des betreffenden Maschinenprogrammabschnittes. Wir schreiben also das LSB der Programmfortführung ($CE) nach $311. Das MSB bleibt unverändert $02.

```
02C8   LDA   #$CE
02CA   STA   $0311
02CD   RTS
```

Mit dem RTS sind wir wieder im Basic-Programm gelandet, welches nun normal weiterverarbeitet wird. Erst ein neues USR-Kommando - im Programm oder im Direktmodus - startet den zweiten Teil unseres Maschinenprogrammes (weil in $0311, - der Einsprungpunkt des USR-Befehls - die Startadresse der auszuführenden Routine steht).

In diesem zweiten Teil müssen wir erst einige Befehle geben, die Sie jetzt vielleicht noch nicht verstehen. Das hängt damit zusammen, daß zum Herauslesen des RAM unter dem ROM das ROM ausgeschaltet werden muß (entspricht POKE 1,53):

```
02CE   LDA   $01
02D0   PHA
02D1   LDA   #$35
02D3   STA   $01
```

(Der PHA-Befehl dient hier zur Zwischenspeicherung des Akku-Inhaltes). Das ist hiermit geschehen und wir kommen wieder in bekannte Gefilde mit der Ausleseschleife:

```
02D5   LDX   #$27
02D7   LDA   $E000,X
02DA   LDY   $E028,X
02DD   STA   $0400,X
02E0   TYA
02E1   STA   $D800,X
02E4   DEX
02E5   BPL   $02D7
```

Damit ist die gesamte gespeicherte Kopfzeile wieder zurückgeholt und wir können das ROM wieder einschalten:

```
02E7   PLA
02E8   STA   $01
```

Falls nun wieder ein USR-Kommando auftaucht, soll die Kopfzeile mit dem 1. Programmteil unter das obere ROM gelegt werden wie am Anfang. Wir müssen deshalb den USR-Vektor auf $02B6 zurückschreiben:

```
02EA   LDA   #$B6
02EC   STA   $0311
02EF   RTS
```

Das wärs! Wenn nun im Programm oder im Direktmodus wieder ein USR-Befehl auftritt, kann das Ganze von vorne beginnen. In dieser Version wird jedesmal eine neue Kopfzeile hin- und wieder zurückgeschoben. Wenn Sie eine einmal festgelegte Kopfzeile immer wieder benutzen möchten, dann stellen Sie den USR-Vektor einfach nicht mehr zurück: Lassen Sie also die Befehle bei 02EA und 02EC weg. Das Programm endet in dem Fall mit:

```
02EA   RTS
```

Eine wichtige Bemerkung noch: So bequem der Ort auch ist, an dem unser kurzes Programm steht, er hat einen gravierenden Nachteil: Falls Sie mittels einer RESET-Taste oder per Software einen Basic-Kaltstart durchführen, geht unser Programm flöten! Dieser Speicherbereich wird im Reset-Programm nämlich mit lauter Nullen überschrieben. Deswegen speichern Sie es bitte bald ab.

In Bild 18 finden Sie ein kleines Testprogramm für unsere Verschieberoutine, und in Tabelle 8 eine Zusammenfassung aller wichtigen Daten der neuen Befehle.

33. Wir stapeln

In Kapitel 28 haben wir beim JSR-Befehl schon den Stapel etwas kennengelernt. Aber so ganz genau wissen wir's ja noch nicht, was das ist. Deswegen jetzt mal im Detail: Der Stapel, auch Prozessorstack genannt, ist der Speicherbereich von dezimal 256 ($100) bis dezimal 511 ($1FF), der direkt von unserer CPU verwaltet wird. Das ist also die gesamte Page 1. Ähnlich wie bei der String-Verwaltung geschieht auch hier das Füllen von oben nach unten. Das erste Byte, welches in den Stack geschoben wird, kommt also nach $1FF, das nächste nach $1FE und so weiter. Voll ist der Stapel, wenn auch $100 besetzt wurde (siehe Bild 19).

```
1 REM ***********************************    <250>
2 REM *                                 *    <229>
3 REM *  TEST FUER DIE 1. VERSION DES *      <139>
4 REM *       PROGRAMM-PROJEKTES        *    <048>
5 REM * V E R S C H I E B E N   V O N   *    <009>
6 REM *       SPEICHERBEREICHEN         *    <193>
7 REM *                                 *    <234>
8 REM * HEIMO PONNATH    HAMBURG 1984 *      <081>
9 REM ***********************************    <002>
10 REM                                       <153>
15 REM ++++++ USR-VEKTOR EINSTELLEN ++++     <065>
20 REM                                       <163>
25 POKE 785,182:POKE 786,2                   <239>
30 REM                                       <173>
35 REM ++++++ KOPFZEILE ++++++++++++++++     <013>
40 REM                                       <183>
45 PRINT CHR$(147)CHR$(18)"TEST
   : BILD $0400=1024, FARBE $D800=55296"CHR$(14
   6)                                        <071>
50 PRINT:PRINT:PRINT"DURCH IRGENDEIN USR-KOMMAN
   DO WIRD NUN IM PROGRAMM-MODUS"            <020>
55 PRINT"DER ERSTE TEIL DES VERSCHIEBE-PROGRAMM
   ES AUFGERUFEN"                            <110>
60 PRINT"DIE KOPFZEILE WIRD UNTER DAS OBERE
    ROM{2SPACE}KOPIERT."                     <215>
65 REM                                       <208>
70 REM ++++++ 1. USR-AUFRUF ++++++++++++     <042>
75 REM                                       <218>
80 A=USR(1)                                  <124>
85 PRINT:PRINT"HIER GESCHIEHT DAS DURCH A=USR(1
   ) IN{4SPACE}ZEILE 65"                     <132>
90 PRINT"DABEI IST 1 EIN DUMMY UND MIT A FANGEN
   {2SPACE}WIR AUCH NICHTS WEITER AN."       <063>
95 PRINT"AUF TASTENDRUCK WIRD DER BILDSCHIRM
   {2SPACE}GE-LOESCHT"                       <029>
100 REM                                      <243>
105 REM ++UEBERSCHREIBEN DER KOPFZEILE ++    <046>
110 REM                                      <253>
115 POKE 198,0:WAIT 198,1:PRINT CHR$(147)    <090>
120 REM                                      <007>
125 REM +++ NEUBEGINN DES PROGRAMMES ++++    <173>
130 REM                                      <017>
135 PRINT CHR$(19)"WAS AUCH IMMER JETZT IN DER
    KOPFZEILE{3SPACE}STEHT, ES WIRD BEIM 2.USR"
                      <017>
140 PRINT"VON DEM ZUVOR DURCH DAS ERSTE USR
    GE-{3SPACE}SPEICHERTE UEBERSCHRIEBEN"    <078>
145 PRINT:PRINT"WENN SIE JETZT EINE TASTE DRUEC
   KEN..."                                   <104>
150 POKE 198,0:WAIT 198,1                    <246>
155 REM                                      <042>
160 REM ++++++ 2. USR-AUFRUF +++++++++++     <090>
165 REM                                      <052>
170 A=USR(1):PRINT                           <169>
175 PRINT"IST DIE ALTE KOPFZEILE ZURUECK IN
    DEN{3SPACE}BILDSCHIRMSPEICHER GESCHOBEN."
                      <164>
180 END                                      <052>
```

Bild 18. Test und Demonstration der Verschieberoutine. Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm mittels eines Programmaufrufes über die USR-Funktionen.

Warum heißt das Ding nun eigentlich Stapel? Das erklärt sich aus dem Zugriffs-Prinzip. Man spricht von einer LIFO-Struktur, von »Last In – First Out«, zu deutsch »zuletzt hinein – zuerst heraus«. Das zuerst hineingebrachte Byte befindet sich am Speicherboden ($1FF), das zuletzt eingebrachte an der Speicherspitze. Stellen Sie sich einen Stapel Akten vor (Bild 20).

Offensichtlich wurde der 4. Aktenordner zuletzt auf den Stapel gesteckt. Er kann zuerst heruntergeholt werden. An die Akte 1 kommen wir erst heran, wenn alle anderen heruntergenommen worden sind. Genauso verhält es sich mit dem Prozessorstack: Um an das unterste Byte des Stapels heranzukommen, müssen erst Byte für Byte die darüberliegenden (nach Bild 19 eigentlich die darunterliegenden) weggeschafft werden.

Mit dem Prinzip des Stapelspeichers werden Sie sich auskennen, wenn Sie schon mal andere Programmiersprachen als Basic ausprobiert haben: In Forth beispielsweise operieren Sie ständig mit Stapeln.

Damit wir – und der Prozessor – den Überblick über den Stack behalten, gibt es dankenswerterweise noch einen Stapelzeiger (stackpointer), der jeweils auf den nächsten freien Platz des Stapels weist. Da gibt's nun aber ein kleines Problem: Der Stapel belegt die komplette Seite 1.

Ein Stapelzeiger, der auf zum Beispiel $01FE zeigen soll, müßte das MSB (also 01) und das LSB (also FE) in zwei Bytes lagern. Der Stapelzeiger ist aber nur 8 Bit groß ... Freundlicherweise sorgt unser Mikroprozessor automatisch für das neunte Bit. Der Zeiger zählt also immer von $FF an rückwärts bis $00 und weist dabei von $1FF bis $100.

Der Stack hat in unserem Computer drei Aufgaben zu erfüllen:

1) Organisation von Unterprogramm-Adressen
2) Zwischenspeicherung bei Unterbrechungen (Interrupts)
3) vorübergehende Datenspeicherung

Die Rolle des Stapels bei Unterprogramm-Aufrufen haben wir in der letzten Folge schon ausgiebig behandelt. Die

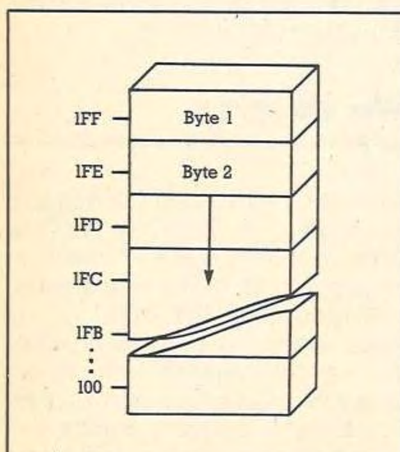


Bild 19. So wird der Stapel gefüllt

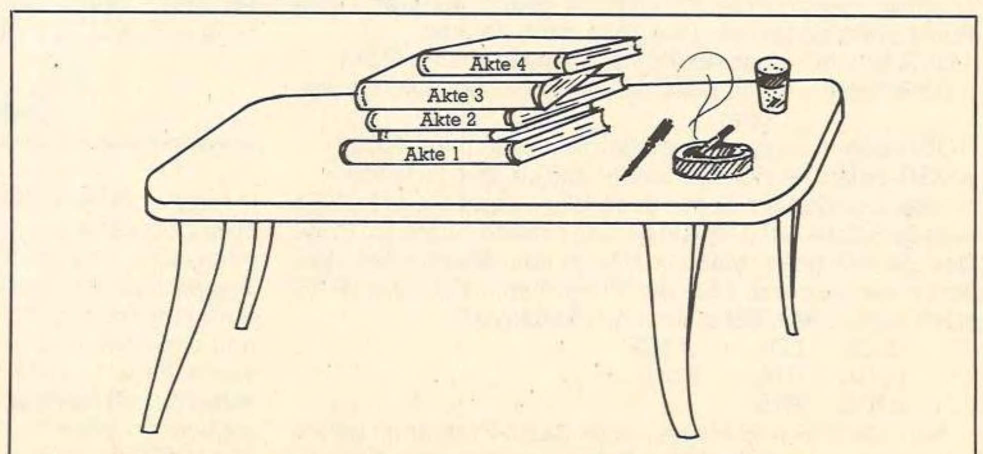


Bild 20. Der Aktenstapel verdeutlicht das Prinzip des Prozessorstacks

sogenannten Interrupts heben wir uns noch für später auf - dazu fehlen uns noch ein paar Kenntnisse. Mit der vorübergehenden Speicherung von Daten befassen wir uns gleich, wenn wir an die Befehle zur Stackbehandlung herangehen.

Zuvor - weil das hier gerade ganz gut paßt - noch ein paar Gedanken zur rekursiven Programmierung. Gemeint ist damit eine Programmstruktur, in der sich ein Unterprogramm selbst aufruft. Auch GOSUB-Befehle in Basic bewirken Einträge der Rücksprungadressen im Stapel. Auf diese Weise ergibt sich für unseren Computer eine begrenzte Verschachtelungstiefe bei Unterprogrammaufrufen. Diese wird bei Rekursion besonders schnell erreicht, und das bewirkt die Ausgabe einer OUT OF MEMORY-Fehlermeldung.

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Dez	Takt-zyklen	Beeinflussung von Flaggen
LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
	absolut,Y	3	B9	185	4	N,Z
LDX	absolut,Y	3	BE	190	4	N,Z
	0-page-abs,Y	2	B6	182	4	N,Z
LDY	absolut,X	3	BC	188	4	N,Z
	0-page-abs,X	2	B4	180	4	N,Z
STA	absolut,X	3	9D	157	5	/
	absolut,Y	3	99	153	5	/
	0-page-abs,X	2	95	149	4	/
STX	0-page-abs,Y	2	96	150	4	/
STY	0-page-abs,X	2	94	148	4	/
INC	absolut,X	3	FE	254	7	N,Z
	0-page-abs,X	2	F6	246	6	N,Z
DEC	absolut,X	3	DE	222	7	N,Z
	0-page-abs,X	2	D6	214	6	N,Z
ADC	absolut,X	3	7D	125	4	N,V,Z,C
	absolut,Y	3	79	121	4	N,V,Z,C
	0-page-abs,X	2	75	117	4	N,V,Z,C
SBC	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
	0-page-abs,X	2	F5	245	4	N,V,Z,C
CMP	absolut,X	3	DD	221	4	N,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
	0-page-abs,X	2	D5	213	4	N,Z,C
BIT	absolut	3	2C	44	4	N,V,Z
	0-page-abs.	2	24	36	3	N,V,Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N,Z
TAY	implizit	1	A8	168	2	N,Z
TXA	implizit	1	8A	138	2	N,Z
TYA	implizit	1	98	152	2	N,Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Tabelle 8. Zusammenfassung aller wichtigen Daten der neuen Befehle

34. Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS

Mit dem Stapel haben wir 256 Speicherplätze für eine schnelle Zwischenspeicherung aller möglichen Daten zur Verfügung. Weil der 6502 diesen Speicherbereich wie die Zeropage behandelt, geht das Speichern sehr schnell. Man muß nur immer die spezielle LIFO-Struktur berücksichtigen.

Im Grunde braucht man eigentlich nur zwei Befehle: Etwas auf den Stapel schieben (in der Literatur oft als Push-Befehl bezeichnet) und etwas herunterziehen, das nennt man dann Pull- oder auch Pop-Befehl.

Unser Prozessor kennt insgesamt sechs auf den Stapel wirkende Anweisungen:

PHA Damit schreibt man den Akku-Inhalt in den Stapel (»PusH-Accumulator«). Der Stapelzeiger wird automatisch eine Position heruntergezählt (er rechnet ja von $FF an abwärts!). Der Inhalt des Akku wird dabei nicht verändert. Deswegen bleibt auch das Status-Register (also die ganzen Flaggen: N V B D I Z C) unbeeinflußt.

PLA »PuLl Accumulator«. Das ist der umgekehrte Weg: Das, was zuoberst auf dem Stapel liegt, wird in den Akku geschrieben. Dadurch wird ein Stapelplatz frei, was den Stapelzeiger veranlaßt, um 1 zu wachsen. Weil das, was da in den Akku geladen wird, 0 sein kann oder auch negativ (also mit gesetztem Bit 7), wird unter Umständen auch die N- oder die Z-Flagge verändert.

Weniger mit Datenzwischenspeicherung haben die anderen Befehle zur Stapel-Manipulation zu tun:

PHP Das steht für »PusH Processor status«, also »schiebe das Prozessor-Status-Register auf den Stapel«. Der aktuelle Flaggenstand kann damit aufbewahrt werden. Das Status-Byte ändert seinen Inhalt dabei ebensowenig wie der Akku bei PHA. Auch hier wird der Stapelzeiger freundlicherweise um 1 herabgezählt.

PLP »PuLl Processor status«, »hole den Prozessor-Status vom Stapel« ist der umgekehrte Befehl, der (wie bei PLA in den Akku) das, was zuoberst im Stapel liegt, in das Flaggen-Register schreibt. Da sollte man höllisch aufpassen, was man damit einlädt: Das ist eine feine Gelegenheit für den Computer, abzustürzen. Der Stapelzeiger wird - wie gehabt - um 1 erhöht.

Nicht direkt mit dem Stapel, sondern mit dem Stapelzeiger befassen sich die beiden folgenden Befehle:

TSX »Transfer Stack-pointer into X«, zu deutsch, »schiebe den Stapelzeiger ins X-Register« eröffnet die Möglichkeit, den Stapelzeiger zu lesen. Dabei bleibt er selbst unverändert erhalten. Weil nun im X-Register alle Werte zwischen $FF und 0 auftreten können, werden auch die Flaggen beeinflußt (N- und Z-Flagge).

TXS Den umgekehrten Weg geht »Transfer X into Stackpointer« = »übertrage X-Register-Inhalt in den Stapelzeiger«. Das ist der einzige Befehl, der es erlaubt, den Stapelzeiger mit einem von uns kontrollierten Wert zu laden. Der Inhalt des X-Registers bleibt dabei unverändert, demzufolge interessieren sich auch die Flaggen nicht dafür.

Alle sechs Anweisungen bestehen nur aus einem Byte und sind implizit adressiert. Die Stapelzeiger-Befehle TXS und TSX benötigen zwei Taktzyklen, die Push-Befehle je drei und die Pull-Befehle vier Taktzyklen zur Bearbeitung.

Es ist etwas schwierig, Stapel-Operationen direkt zu verfolgen. Die meisten Assembler - so anscheinend auch der SMON - gebrauchen ebenfalls diesen Speicherbereich. Verlangt man beispielsweise mit dem SMON-Kommando M 0100 01FF eine Darstellung des Stapelinhaltes, dann findet man eine ganze Menge Spuren der Arbeit des Assemblers. Versucht man die zu löschen oder zu überschreiben, zum Beispiel mit dem nachfolgenden kleinen Programm, dann hat der Assembler die Mühe schon wieder zunichte gemacht, wie man durch erneutes M 0100 01FF schnell sehen kann. Dieses kleine Programm soll unterhalb des durch den Stapelzeiger bezeichneten Bereichs 32 Nullen in den Stapel schreiben:

```
8000   LDA   #$00
8002   TSX
```

Der Stapelzeiger wird ins X-Register gerettet.

```
8003   LDY   #$20
8005   PHA
```

Wir schieben eine Null auf den Stapel.

```
8006   DEY
8007   BNE   $8005
8009   TXS
```

Nach 32 Eintragungen von Nullen stellen wir den alten Stapelzeiger wieder her.

```
800A   BRK
```

Erneutes Kommando M 0100 01FF zeigt keine Nullen. Erst wenn wir anstelle des TXS in Zeile 8009 ein BRK schreiben, den Stapelzeiger also nicht zurückschreiben, erscheinen unsere Nullen. Sieht man genau hin, dann stellt man fest, daß unterhalb des durch den Stapelzeiger bezeichneten Bereichs genau der gleiche Inhalt zu finden ist wie vorher, nur eben mit dem Stapelzeiger verschoben.

Ganz konnte ich dies Rätsel noch nicht lösen, muß ich gestehen, aber für den Gebrauch des Stapels ändert sich dadurch für uns nichts. Worauf muß man achten bei Stapeloperationen? Ganz einfach: Zwischen dem Ablagern eines Wertes auf dem Stapel und dem Zurückholen muß für jeden Push-Befehl ein Pull-Befehl vorhanden sein, für jedes weitere PHA ein PLA, für jedes JSR ein RTS. Nur wenn wir auf diese Symmetrie der Push- und der Pull-Befehle achten (und wie Sie noch aus der vorhergegangenen Ausgabe wissen, sind ja JSR und RTS ebenfalls dazuzurechnen), können wir sicher sein, daß der Stapelzeiger zum Zeitpunkt des Rückholens eines Wertes vom Stapel auch wirklich darauf deutet. Wenn man also nicht ganz genau weiß, wie der verwendete Assembler den Stapel nutzt, sollte man auf Operationen mit den Befehlen TSX und TXS verzichten.

Nun können Sie schon einen Teil der bislang unbekannten Programmsequenz aus der letzten Folge verstehen. Im zweiten Programmteil hatten wir mit

```
02CE   LDA   $01
02D0   PHA
```

den Inhalt der Speicherstelle $01 in den Akku geladen und auf den Stapel geschoben. Später - nach einigen weiteren Operationen - wurde dann dieser Speicherinhalt wiederhergestellt durch

```
02E7   PLA
02E8   STA   $01
```

Was aber hat es mit dieser Speicherstelle $01 auf sich? Das soll nun als nächstes erklärt werden.

35. Sein oder Nichtsein: Das Rätsel des Prozessorports

Der Commodore 64 hat 64 KByte an RAM zu bieten. Außerdem aber verfügen wir beim normalen Programmieren über weitere 24 KByte, in denen das Betriebssystem, der Basic-Interpreter, Ein- und Ausgabebausteine und der Zeichenspeicher stecken. Wie Sie wissen, umfaßt der Adreßbus aber nur 16 Bit, was uns lediglich 65536 Speicherzellen, also 64 KByte adressieren läßt. Des Rätsels Lösung liegt darin, daß einige Adressenbereiche mehrfach belegt sind. Man kann das vergleichen mit dem Trick des Kastens mit dem doppelten Boden. Welcher Kasteninhalt gerade dem Prozessorzugriff offensteht, wird durch den Prozessorport, das sind die Speicherstellen $00 und $01, gesteuert.

Dr. Helmuth Hauck hat in seiner Serie »Memory Map mit Wandervorschlägen« (64'er, Ausgabe 11 (1984), Seite 135 ff.) die genaue Funktion jedes Bits dieser beiden Speicherstellen erklärt. Wer noch mehr wissen möchte - auch über die Wirkungsweise der beiden Leitungen »Game« und »Exrom« - sollte das nachlesen im »Commodore 64 Programmers Reference Guide« ab Seite 260. Für uns als angehende Assembler-Alchimisten ist die Speicherstelle 1 aber so wichtig, daß wir ganz kurz hier noch mal darauf eingehen.

Die Speichersteuerfunktionen haben die Bits 0 bis 2 der Speicherstelle 1. Je nach Belegung dieser Bits gestaltet sich die 64-KByte-Landschaft unseres Computers wie in Tabelle 9 und in Bild 20a gezeigt.

Was können wir als Maschinen-Programmierer mit dieser Kenntnis anfangen? Theoretisch stehen uns für unsere Programme damit 64 KByte offen. Praktisch werden wir nur in den seltensten Fällen auf die Ein- und Ausgabe-Bausteine verzichten können. Lassen wir ein reines Maschinenprogramm laufen, ohne jeglichen Rückgriff auf Interpreter oder Betriebssystem, dann haben wir immerhin noch zirka 60 KByte zur freien Verfügung.

Benutzen wir Routinen aus diesen beiden ROM-Bausteinen, dann müssen wir sie allerdings - zumindest für den Zeitpunkt des Routineaufrufs - wieder einschalten. Wenn wir - was wohl meistens der Fall sein wird - Kombinationen von Basic- und Assemblersprache verwenden, können wir den gesamten Basic-Speicher bis $A000 frei halten, können auch den bei allen Beispielprogrammen so beliebten Bereich $C000 bis $D000 leer lassen und packen unsere Routinen weitgehend unter die ROMs, die dann jeweils beim Aufruf abgeschaltet werden. So haben wir eine Menge zusätzlichen Speicherplatz ergattert.

Nun können wir auch den letzten Rest des bislang unklaren Programms aus Kapitel 32 verstehen. Nachdem wir den Inhalt der Speicherstelle $1 auf den Stapel gerettet haben (Zeilen $02CE und $02D0), schreiben wir $35 in den Prozessorport:

```
02D1   LDA   #$35
02D3   STA   $01
```

$35 ist binär 0011 0101. Die Bits 0 bis 2, auf die es uns in diesem Zusammenhang ankommt, bewirken nun das Ausschalten des Interpreters und des Betriebssystems. Die Ein- und Ausgabe-Bausteine bleiben aktiv. Im weiteren Programmverlauf lesen wir die Speicherinhalte ab $E000, wobei wir nun den RAM-Inhalt erfassen. Das sollte vielleicht noch mal klargestellt werden: Jedes Hineinschreiben in die mehrfach belegten Speicherbereiche (dabei sind die Ein- und Ausgabe-Bausteine aber ausgenommen) wird automatisch in den RAM-Bereich umgelenkt. Das ist ja auch klar: In ein ROM kann eben nicht geschrieben werden. Deshalb braucht man dabei die ROMs nicht auszuschalten. Jeder Lesevorgang greift aber auf die ROMs zu, weshalb man sie in unserem Fall ausschalten muß. Wie schon oben beim Stapel erklärt, schalten wir durch das Zurückholen des vorher dorthin geretteten alten Inhalts der Speicherstelle $1 in den Prozessorport wieder den Normalzustand ein.

Speicherstelle 1 Bits 2 1 0	$A000-$BFFF	$D000-$DFFF	$E000-$FFFF
1 1 1	Basic	I/O	Kernel
1 1 0	RAM	I/O	Kernel
1 0 1	RAM	I/O	RAM
1 0 0	RAM	RAM	RAM
0 1 1	Basic	Zeichen	Kernel
0 1 0	RAM	Zeichen	Kernel
0 0 1	RAM	Zeichen	RAM
0 0 0	RAM	RAM	RAM

Tabelle 9 zeigt, welche Bausteine bei verschiedener Belegung der Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind.

36. Die indirekte Adressierung

Wir werden nun die beiden letzten noch ausstehenden Arten der Adressierung kennenlernen. Beides sind indirekte Adressierungsarten. Mit dem indirekten JMP-Befehl (zum Beispiel sind wir in Kapitel 28 schon vertraut geworden. Wir

hatten auch gelernt, daß es sich hierbei um einen absoluten Einzelgänger handelt, der nur für so einen Sprung erlaubt ist. Ebenso haben wir die indizierte Adressierung zu beherrschen gelernt: Das war die Sache mit den Indexregistern X oder Y. Eine Kombination aus beiden (also der indirekten und der indizierten) Adressierungsarten sind die indiziert-indirekte und die indirekt-indizierte Adressierung.

Die indirekt-indizierte Adressierung

Fangen wir mit der sehr häufig benutzten indirekt-indizierten Adressierung an: Man nennt sie auch »indirekt Y« oder »nach-indizierte indirekte« Adressierung. Am besten sehen wir uns mal so einen Befehl an:

```
LDA     ($FA),Y
```

Die Klammer erinnert uns an den indirekten JMP-Befehl. Tatsächlich hat sie hier auch dieselbe Funktion: In $FA und $FB steht ein Zeiger auf eine Adresse. Nehmen wir mal an, die Belegung der Speicher wäre:

```
$FA     $01
$FB     $80
```

und im Y-Register stünde eine 5. Der Zeiger $FA/FB weist also auf die Speicherstelle $8001. Da haben wir also wieder das Prinzip des toten Briefkastens. Der Computer guckt in den hohlen Baum $FA/FB (LSB in $FA, MSB in $FB) und findet dort die Treffpunktadresse. Nun sind diese toten Briefkästen aber auch den gegnerischen Alchimisten-Agenten bekannt. Es kommt also noch ein Trick dazu: Zur dort aufgefundenen Adresse wird der Inhalt des Y-Registers addiert.

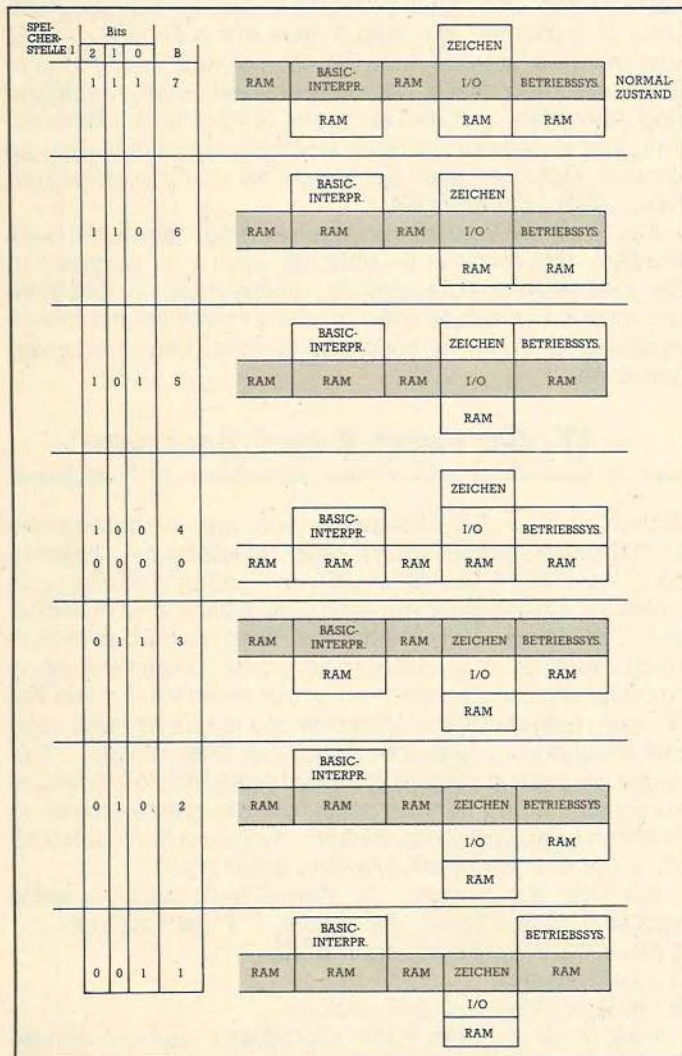


Bild 20a. So wirken sich Veränderungen der Bits 0 bis 2 von Speicherstelle 1 auf dem Speicherzustand aus

In unserem Fall fanden wir also in $FA/FB die Adresse $8001, im Y-Register steht eine 5, somit ist die endgültige Adresse $8001+5 = $8006. Unser Beispiel »LDA(FA),Y« bewirkt daher, daß in den Akku der Inhalt der Speicherstelle $8006 geladen wird. Nachindiziert nennen manche die Adressierung deswegen, weil zunächst dem Zeiger nachgegangen wird, der in unserem Beispiel auf $8001 weist, und erst danach durch Addition des Inhalts des Y-Registers die endgültige Speicherstelle (hier also $8006) berechnet wird.

Als Zeiger (also die Adresse in der Klammer) sind nur Zeropage-Speicherstellen verwendbar, als Indexregister darf man hier nur das Y-Register gebrauchen. Von den bisher behandelten Befehlen können ADC, CMP, LDA, SBC und STA mit dieser Adressierungsart verwendet werden. Genaueres finden Sie wieder in der Tabelle mit der Befehlsübersicht (Tabelle 10).

Bevor wir uns dem anderen indirekten Adreß-Modus zuwenden, wollen wir uns überlegen, wozu man die indirekt-indizierte Adressierung verwendet. Wie Sie sich natürlich erinnern können, konnte man mit der normalen indizierten Adressierung, zum Beispiel mit

```
LDA $8000,Y
```

durch Variation des Indexregisters (hier das Y-Register) 256 Speicherstellen erfassen (Y von $FF herunter bis 00).

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LDA	indirekt X	2	A1	161	6	N,Z
	indirekt Y	2	B1	177	5*	N,Z
STA	indirekt X	2	81	129	6	—
	indirekt Y	2	91	145	6	—
ADC	indirekt X	2	61	97	6	N,V,Z,C
	indirket Y	2	71	113	5*	N,V,Z,C
SBC	indirekt X	2	E1	225	6	N,V,Z,C
	indirekt Y	2	F1	241	5*	N,V,Z,C
CMP	indirekt X	2	C1	193	6	N,Z,C
	indirekt Y	2	D1	209	5*	N,Z,C
PHA	implizit	1	48	72	3	—
PLA	implizit	1	68	104	4	N,Z
PHP	implizit	1	08	8	3	—
PLP	implizit	1	28	40	4	alle
TSX	implizit	1	BA	186	2	N,Z
TXS	implizit	1	9A	154	2	—

* Wenn bei der Befehlsausführung eine Page-Grenze überschritten wird, muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 10. Übersicht der in dieser Folge vorgestellten Befehle

Will man mehr als diese 256 berücksichtigen, dann muß eine neue Basis (im Beispiel also anstelle der $8000) gewählt werden. Um das zu illustrieren, sehen wir uns mal den Anfang eines Programms an, welches den gesamten Bildschirminhalt ausliest und nach $E000 schreibt:

```
1000  LDY   #$00
1002  LDA   $0400,Y
1005  STA   $E000,Y
1008  LDA   $0500,Y
100B  STA   $E100,Y
100E  LDA   $0600,Y
1011  STA   $E200,Y
1014  LDA   $0700,Y
1017  STA   $E300,Y
101A  DEY
101B  BNE   $1002
...
```

Wie Sie sehen, erfordert das durch die Tatsache, daß vier Blöcke zu je 256 Byte übertragen werden müssen, immer-

hin schon 28 Byte Programmtext. Nun soll die indirekt-indizierte Adressierung verwendet werden, um dieselbe Aufgabe zu lösen. Wir legen zunächst zwei Zeiger auf der Zeropage fest:
$FA/FB sollen die Bildschirmadresse enthalten
$FC/FD die Zieladresse ab $E000.

```
1000  LDA  #$00
1002  STA  $FA
1004  STA  $FC
```

Das waren die LSBs der Zeiger, es folgen die MSBs:

```
1006  LDA  #$04
1008  STA  $FB
100A  LDA  #$E0
100C  STA  $FD
```

Damit sind die Zeiger festgelegt. Es sind vier Blöcke zu je 256 Byte zu übertragen. Diese Blockanzahl legen wir ins X-Register als Zähler:

```
100E  LDX  #$04
```

Dann laden wir ins Y-Register ebenfalls einen Zähler (den Index):

```
1010  LDY  #$00
```

Jetzt kann die eigentliche Übertragungsschleife starten:

```
1012  LDA  ($FA),Y
1014  STA  ($FC),Y
1016  DEY
1017  BNE  $1012
```

Wenn das Y-Register wieder bei 0 angekommen ist (von der ersten 0 nach einem Unterlauf über $FF, $FE und so weiter bis 0), ist der erste Block übertragen. Wir erhöhen nun das MSB beider Zeiger um 1:

```
1019  INC  $FB
101B  INC  $FD
```

Außerdem zählen wir den Blockzähler um 1 herunter:

```
101D  DEX
101E  BNE  $1012
...
```

Wenn das Programm auf diese Weise auch drei Byte mehr Speicherplatz braucht, ist doch leicht der Vorteil zu sehen: Müssen wir nämlich (statt nur vier) mehr Blöcke übertragen (bis zu 255), dann verändert sich unser zweites Programm um keinen Deut (außer dem Zähler im X-Register, der nun mit der jeweils anderen Block-Anzahl geladen wird), während die erste Programmtechnik für jeden weiteren Block um sechs Bytes erweitert werden muß.

Es gibt noch eine ganze Reihe von Anwendungsmöglichkeiten, die die indirekt-indizierte Adressierung so attraktiv machen. Für Geschwindigkeitsfanatiker (ich selbst bin bei Grafik-Fragen auch einer!) muß aber gesagt werden, daß dem Speicherplatzvorteil ein Geschwindigkeitsnachteil gegenübersteht. Jeder indirekt-indiziert adressierte Befehl braucht einen Taktzyklus länger als der vergleichbare absolut-indizierte Befehl. Zu diesen Feinheiten werden wir aber später noch kommen.

Die indiziert-indirekte Adressierung

Wenden wir uns nun der letzten noch fehlenden Adressierungsart zu, der indiziert-indirekten. Man nennt sie auch »vorindizierte indirekte« oder »indirekt X« Adressierung. Sehen wir auch hier zunächst ein Beispiel an:

```
STA  ($FA,X)
```

Auch hier drückt die Klammer wieder aus, daß der Klammerinhalt ein Zeiger ist. Das ist jetzt aber nicht das Bytepaar $FA/FB, sondern zur angegebenen Adresse $FA soll noch der Inhalt des X-Registers addiert werden. Nehmen wir mal an, dort stünde eine 2, dann wird der Zeiger $FC/FD mit diesem Befehl angesprochen, denn $FA+2=$FC und entsprechend $FB+2=$FD. Wenn in den Speicherstellen $FA bis $FF folgender Inhalt zu finden ist:

```
00FA  $00
00FB  $04     FA/FB = $0400
00FC  $00
00FD  E0      $FC/FD = $E000
00FE  $10
00FF  $80     FE/FF = $8010
```

dann könnte das eine ganze Tabelle von Zeigern sein, die jeweils durch den X-Registerinhalt angesprochen werden. Der Akkuinhalt wird in unserem Beispiel nach $0400 geschrieben, wenn im X-Register 0 steht, nach $E000, wenn das X-Register eine 1 enthält und nach $8010, wenn stattdessen eine 2 im X-Register zu finden ist.

Sie werden sich vielleicht auch bei diesem Beispiel gefragt haben, was passiert, wenn im X-Register unseres Beispiels eine 6 steht. Nun, unser 8-Bit-Prozessor läuft über, und wir finden einen Zeiger $00/01.

Rein theoretisch ist diese Adressierungsweise ganz interessant. Aber auf der Zeropage ist's reichlich eng, und nur selten kommt man daher in die Lage, dort eine Zeigertabelle einzurichten, die man mittels des X-Registerinhalts und der indiziert-indirekten Adressierung abgreifen kann. Die Bedeutung dieser Adressierungsart ist also recht gering. Außerdem erfordert sie sechs Taktzyklen zur Bearbeitung und ist somit auch noch langsam. Von den bisher bekannten Befehlen sind die folgenden damit verwendbar: ADC, CMP, LDA und STA.

Bevor wir die Adressierung zu den Akten legen, sei noch erwähnt, daß manche Lehrbücher noch eine weitere Art, die Akkumulator-Adressierung, unterscheiden. Betroffen sind davon vier 1-Byte-Befehle, die wir noch kennenlernen werden und die man ebensogut als implizit adressiert ansehen kann.

37. Die ersten Kernel-Routinen

Sicher werden Sie alle schon von der Kernel-Routine $FFD2 gehört haben und sie vielleicht auch schon verwenden. Wenn nicht, um so besser, denn dann sind Sie noch nicht vom einseitigen Gebrauch dieses Instruments verdorben. Die meisten Kernel-Adressen sind nämlich sehr vielseitig verwendbar, je nach den Vorgaben. Das ist wie mit einem Haushaltsgerät, das immer nur zum Rühren von Kuchenteig eingesetzt wird. Dabei kann man damit auch noch Saft machen, Gurken schnitzeln, Getränke mixen ... Genauso wie man in diesem etwas schiefen Vergleich die Gebrauchsanleitung kennen sollte, um die ganzen anderen Funktionen ausnutzen zu können, muß man hier noch einige Dinge über die Kernel-Aufrufe beherzigen.

Für jede Verwendung der Kernel-Sprungtabelle sollte man sich angewöhnen, dies in drei Schritten zu tun:
1) die nötigen Vorbereitungen treffen,
2) Routineaufruf,
3) Fehlerabfrage und -behandlung.

Fangen wir mit dem Punkt »Vorbereitungen« an. Einige Routinen brauchen Informationen, die ihnen erst durch andere Kernel-Routinen beschafft werden. Ruft man diese anderen Routinen vorher nicht auf, dann funktioniert auch der erwünschte Aufruf nicht richtig. Wenn die Routine einen bestimmten Wert im Y-Register erwartet, dann muß der dort auch stehen. Wenn nicht, dann geht das Programm »in die Hose«. Bei jeder Kernel-Routine, die hier beschrieben wird, gebe ich alle nötigen Vorbereitungen an.

Der Routinenaufruf sollte immer mittels JSR erfolgen. Alle auf diese Weise aus der Kernel-Sprungtabelle abzurufenden Programme enden nämlich mit einem RTS. Damit keine wichtigen Werte aus dem Aufrufprogramm überschrieben werden, man sie also vor dem Aufruf der Kernel-Routine irgendwohin retten kann, gebe ich auch noch an, welche Register durch die Routine verändert werden und wieviel Stapelspeicherplatz bereitgehalten werden muß.

Die Routinen sind so konstruiert, daß beim Auftreten eines Fehlers nach der Rückkehr das Carry-Bit gesetzt ist. Durch Untersuchen des Carry können so Fehler rechtzeitig erkannt und behandelt werden. Im Akku findet man in dem Fall dann eine Fehlernummer. Die Ausgabe der Fehlermeldung erfolgt also nicht - wie im Basic - in Klarschrift. In Tabelle 9 sind die Fehlernummern und ihre Bedeutung aufgelistet.

Nummer	Text	Bedeutung
0	BREAK	Während des Programms wurde die RUN/STOP-Taste gedrückt
1	TOO MANY FILES	Man kann maximal 10 offene Files einrichten
2	FILE OPEN	Ein bereits geöffnetes File wird nochmals geöffnet
3	FILE NOT OPEN	Auf ein noch nicht geöffnetes File sollte zugegriffen werden
4	FILE NOT FOUND	Das geforderte File ist nicht verfügbar
5	DEVICE NOT PRESENT	Das angesprochene Gerät zeigt keine Reaktion
6	NOT INPUT FILE	Aus einem Schreibfile kann nicht gelesen werden
7	NOT OUTPUT FILE	In ein Lesefile kann nicht geschrieben werden
8	MISSING FILE NAME	Bei Operationen, die einen Filenamen erfordern, fehlt dieser
9	ILLEGAL DEVICE NUMBER	Das versuchte Kommando ist beim angesprochenen Gerät nicht möglich

Tabelle 11. Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku.

Welche Fehlernummern eine Routine ausgeben kann, wird ebenfalls von mir bei jeder Routinen-Besprechung angegeben.

Nun aber zur ersten Routine FFD2, die wie einen Rattenschwanz eine Reihe weiterer nach sich zieht:

Name	CHROUT
Zweck	Ausgabe eines Zeichens
Adresse	$FFD2 dez. 65490
Vorbereitungen	(CHKOUT,OPEN) Zeichen im Akku
Fehler	0
Stapelbedarf	8
Register	Akku

Falls Sie diese Routine schon einmal benutzt haben, dann geschah es vermutlich ohne die Vorbereitungen CHKOUT und OPEN. Freundlicherweise hat unser Computer einige Voreinstellungen schon für uns getroffen. Denn normalerweise sendet CHROUT ein Zeichen über einen schon geöffneten Ausgabekanal, und der ist zum Bildschirm geschaltet. Ein kleines Beispielprogramm soll das illustrieren. Zunächst laden Sie bitte den SMON ein und starten Sie ihn. Nun soll eine Texttabelle angelegt werden. Das funktioniert beim SMON am bequemsten über das K-Kommando. Geben Sie ein K 6000. Der SMON antwortet mit:

'6000 ....

Wenn Sie nun die RUN/STOP-Taste drücken, können Sie mit dem Cursor in diese Punktzeile fahren und einen Text schreiben:

'6000 HALLO ASSEMBLER-ALCHIMIST ....

Sinnvoll - vor allem für die weitere Verwendung dieses Textes - ist es, ein (RETURN), also dezimal 13 oder $0D anzuschließen. Dazu gibt es natürlich den Weg über den Assemblerbefehl. Zur Übung wollen wir aber das M-Kommando verwenden. Geben Sie ein (zuerst die »RETURN«-Taste betätigen) M6018, dann wieder RUN/STOP, und fahren Sie mit dem Cursor auf Speicherstelle 601A (falls Sie in 6019 kein Leerzeichen $20 stehen haben, dann fügen Sie's jetzt noch ein). Geben Sie nun anstelle des dort stehenden Bytes 0D ein, und drücken Sie die RETURN-Taste. Der Monitor sollte jetzt zeigen:

```
:6018 54 20 0D
```

etc.

Unser Text soll mit einem BRK enden. Deshalb gehen wir jetzt in den Assembler-Modus mit dem SMON-Kommando A 601B und schreiben:

```
601B   BRK
```

Nun folgt das eigentliche Prográmmchen, das Byte für Byte bis zur Null (BRK) den Text aus der gerade erstellten Texttabelle liest und mittels FFD2 auf den Bildschirm bringt:

```
601C   LDY   #$00
601E   LDA   $6000,Y
6021   BEQ   $602C
```

Das Y-Register wird als Index initialisiert, dann die Texttabelle in den Akku geladen. Wenn das Programm dabei auf die Null stößt, verzweigt es zum Ende. Jetzt folgt die Routine zur Bildschirmausgabe:

```
6023   JSR   $FFD2
6026   BCS   $602D
```

Falls bei der Kernel-Routine etwas schiefgelaufen ist, wird das Carry-Bit gesetzt, was wir überprüfen und zu einem BRK-Kommando verzweigen (das ist natürlich nur sinnvoll, solange ein Monitor oder Assembler wie der SMON aktiv ist). Nun erhöhen wir das Index-Register und das Ganze beginnt von vorne:

```
6028   INY
6029   JMP   $601E
602C   RTS
602D   BRK
```

Wenn wir nun aus dem SMON mit F und anschließendem X aussteigen und ein kleines Basic-Aufrufprogramm machen (Bei OUT OF MEMORY ERROR bitte NEW eingeben):

```
10 PRINTCHR$(147)
20 SYS 24604 :REM = $601C
30 END
```

dann können wir uns die Wirkung unseres Programms ansehen: Nach RUN wird der Bildschirm gelöscht und unser Text ausgedruckt.

$FFD2 nimmt uns also eine Menge Arbeit ab: Automatisch legt diese Routine in den Bildschirmspeicher den Bildschirmcode (sie rechnet also auch gleich ASCII, das wir ja eingegeben haben, in den POKE-Code um) und in die dazugehörige Bildschirmfarbspeicherstelle den aktuellen Farbcode. Sie setzt außerdem noch den Cursor weiter.

Mit $FFD2 kann man aber noch viel mehr machen! Schließlich ist ja der Bildschirm (Gerätenummer 3) nicht der einzige mögliche Empfänger. Wir wollen als nächstes mal eine Ausgabe mittels $FFD2 auf den Drucker erzielen. Hier sind die Vorbereitungen allerdings nötig. Zunächst mal müssen wir uns noch zwei weitere Kernal-Routinen ansehen, nämlich CHKOUT und OPEN.

Name	CHKOUT
Zweck	Kanal zum Ausgang definieren
Adresse	$FFC9 dez. 65481
Vorbereitungen	OPEN log. Filenummer ins X-Register
Fehler	0,3,5,7
Stapelbedarf	4
Register	Akku, X-Register

Mit dieser Routine kann jedes File, der zuvor durch OPEN spezifiziert worden ist, zum Ausgabe-File erklärt werden. Natürlich muß dann das derart angesprochene Gerät auch ein Ausgabegerät sein. Andernfalls entsteht ein Fehler. Bevor man Daten über einen Kanal senden will, muß CHKOUT durchgeführt werden. Wenn die mittels OPEN übergebene Geräteadresse größer als 3 ist, sendet diese

Routine automatisch auch ein LISTEN-Kommando an das Ausgabegerät. LISTEN setzt dann zum Beispiel den Drucker in Empfangsbereitschaft. Die Durchführung von CHKOUT ist einfach (vorausgesetzt, man hat vorher OPEN aufgerufen): In das X-Register wird die logische Filenummer geschrieben und dann per JSR $FFC9 (CHKOUT) angesteuert.

Nun zur anderen Vorbereitung von $FFD2, zu OPEN.

Name	OPEN
Zweck	Öffnen eines logischen Files
Adresse	$FFC0 dez. 65472
Vorbereitungen	SETLFS,SETNAM
Fehler	1,2,4,5,6
Register	Akku, X- und Y-Register

Die Routine OPEN an sich anzusprechen ist relativ einfach. Es genügt ein JSR $FFC0. Zuvor allerdings - der Rattenschwanz wird länger - muß mit SETNAM der Filename und mit SETLFS die logische Filenummer, die Geräteadresse und eventuell eine Sekundäradresse festgelegt sein. Erst danach kann das File geöffnet werden durch OPEN. Also sehen wir uns noch SETLFS und SETNAM an:

Name	SETLFS
Zweck	Spezifikationen eines logischen Files
Adresse	$FFBA dez. 65466
Vorbereitungen	logische Filenummer in Akku Gerätenummer ins X-Register Sekundäradresse ins Y-Register
Fehler	keine
Stapelbedarf	2
Register	keine

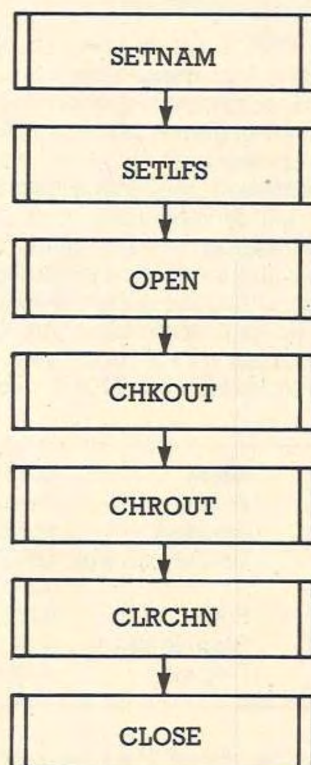


Bild 21. Die Abfolge der Routineaufrufe

SETLFS legt für die anderen Kernel-Routinen logische Filenummer, Gerätenummer und Sekundäradresse fest. Die logische Filenummer ist dabei eine Schlüsselzahl, die in eine durch OPEN angelegte File-Tabelle weist. Die Gerätenummer kann zwischen 0 und 31 liegen, dabei sind folgende Zuordnungen vorgesehen:

0 Tastatur
1 Datasette
2 RS232C-Kanal
3 Bildschirm

Gerätenummern ab 4 beziehen sich automatisch auf Geräte am seriellen Bus. Dabei gilt im allgemeinen:

4 Drucker
8 Diskettenstation

Die Sekundäradresse ist eine Kommandonummer, die für das jeweils angesprochene Gerät spezifisch ist, zum Beispiel 10 bewirkt beim Drucker Commodore 1526, daß das Gerät in die Grundstellung geht (siehe jeweiliges Handbuch). Will man keine Sekundäradresse verwenden, dann muß $FF ins Y-Register geschrieben werden. Der Aufruf von SETLFS geschieht also in folgender Weise: In den Akku lädt man die gewünschte logische Filenummer, ins X-Register die Geräteadresse und ins Y-Register $FF oder aber die Sekundäradresse. Danach erfolgt der Sprung mit JSR $FFBA.

Schließlich noch zu SETNAM:

Name	SETNAM
Zweck	Filenamen festlegen
Adresse	FFBD dez. 65469
Vorbereitungen	Namenslänge in den Akku, LSB des Namenstextes in X-Register, MSB des Namenstextes in Y-Register
Fehler	keine
Stapelbedarf	2
Register	Akku, X- und Y-Register

Vor der Eröffnung eines Files mittels OPEN muß diese Routine den Filenamen festlegen. Dazu schreibt man in den Akku die Länge des Namens und in die Register X, Y die Startadresse (LSB ins X-Register, MSB ins Y-Register) der Namenstext-Tabelle. Der Ort dieser Tabelle ist frei wählbar. Wird kein Filename gewünscht, dann gibt man dem Akku die Länge 0 an. X- und Y-Register sind in dem Fall ohne Bedeutung.

Damit - sollte man meinen - hätten wir nun alle Bedingungen erfüllt, FFD2 zur Ausgabe auf den Drucker zu bewegen. Leider ist das noch nicht der Fall: FFD2 schließt nämlich das File und den Ausgabekanal nicht. Das kann - wenn man's nicht beachtet - zu Fehlern oder zur weiteren Ansprache des Druckers führen, auch wenn die gar nicht mehr erwünscht ist. Deswegen sollten noch zwei Kernel-Routinen angehängt werden, von denen die eine (CLRCHN) alle Ein- und Ausgabekanäle wieder in den Ausgangszustand zurückführt, und die andere (CLOSE) das File ordnungsgemäß schließt:

Name	CLRCHN
Zweck	Ein- und Ausgabekanäle in Ausgangsstellung bringen
Adresse	$FFCC dez. 65484
Vorbereitung	keine
Fehler	keine
Stapelbedarf	9
Register	Akku, X-Register

Der Aufruf von CLRCHN erfolgt einfach durch JSR $FFCC. Die Wirkung ist enorm: Mit einem Schlag werden alle Kanäle freigeräumt. Eingangskanälen wird ein UNTALK (dem Gerät wird gesagt: Halt den Mund), Ausgangskanälen ein UNLISTEN (das bedeutet soviel wie: Hör nicht mehr zu) übermittelt. Der Ausgangszustand stellt sich wieder her: Tastatur als Eingabe-, Bildschirm als Ausgabegerät.

Die endgültig letzte Routine für diesmal ist CLOSE:

Name	CLOSE
Zweck	Schließen logischer Files
Adresse	$FFC3 dez. 65475
Vorbereitungen	logische Filenummer in Akku
Fehler	0
Stapelbedarf	2
Register	Akku, X- und Y-Register

Wenn für ein File alle Ein- und Ausgabeoperationen beendet sind, kann es - nach Einschreiben der Filenummer in den Akku - mittels CLOSE ordnungsgemäß geschlossen

werden. Der Eintrag in der File-Tabelle wird auf diese Weise gelöscht.

So, jetzt sind wir soweit, daß wir die Textausgabe auf dem Drucker programmieren können. Bild 21 faßt die einzelnen Schritte noch mal zusammen.

Und hier das Programm dazu. Wir verwenden die im anderen Beispiel schon aufgebaute Texttabelle weiter. Zunächst also SETNAM:

```
601C  LDA  #$00
601E  JSR  $FFBD
6021  BCS  $6053
```

Wenn ein Fehler aufgetreten ist, findet man ein gesetztes Carry-Bit. In dem Fall wird verzweigt zu einem BRK-Kommando (was die Anwesenheit eines Monitors erforderlich macht, solange man sich noch nicht sicher ist, ob Fehlermeldungen auftauchen). Die Null im Akku besagt, daß kein Filename gewünscht ist. Dann kommt SETLFS:

```
6023  LDA  #$04
6025  LDX  #$04
6027  LDY  #$FF
6029  JSR  $FFBA
602C  BCS  $6053
```

Es wurde ein File festgelegt mit der logischen Filenummer 4, der Geräteadresse 4 und ohne Sekundäradresse. Jetzt geben wir das OPEN-Kommando:

```
602E  JSR  $FFC0
6031  BCS  $6053
```

Der Ausgabekanal wird definiert mit CHKOUT:

```
6033  LDX  #$04
6035  JSR  $FFC9
6038  BCS  $6053
```

Damit sind alle Vorbereitungen erledigt und die Zeichenausgabe kann wie im ersten Programm durchgeführt werden mit CHROUT:

```
603A  LDY  #$00
603C  LDA  $6000,Y
603F  BEQ  $604A
6041  JSR  $FFD2
6044  BCS  $6053
6046  INY
6047  JMP  $603C
```

Alle Zeichen sind nun ausgedruckt. Wir rufen CLRCHN auf:

```
604A  JSR  $FFCC
```

Als letzte Routine folgt nun noch CLOSE:

```
604D  LDA  #$04
604F  JSR  $FFC3
6052  RTS
```

Damit wurde das File Nummer 4 geschlossen. Anschließend erfolgte der Rücksprung aus dem Programm. Für die Fehlerbehandlung habe ich nur einen BRK vorgesehen, der sofortigen Registerüberblick erlaubt, wenn zum Beispiel der SMON im Speicher ist.

```
6053  BRK
```

Ohne Monitor im Speicher kann der Computer allerdings abstürzen oder im besten Fall einen Basic-Warmstart durchführen. Wenn Sie sowas also für Ihre Zwecke programmieren möchten, sollten Sie einen anderen Weg suchen, die Fehler aufzufangen. Man hat ja nicht immer einen Monitor in den Speicher geladen.

Mit diesen sieben Kernel-Routinen beenden wir dieses Kapitel. In unseren Sonderheft 25 haben B. Schneider und K. Schramm in ihrem Kurs »In die Geheimnisse der Floppy eingetaucht« gezeigt, wie man mittels der besprochenen Routinen, und einiger anderer, auch die Diskettenstation ansprechen oder sogar Floppy und Drucker zum »Spooling« veranlassen kann. Das habe ich zwar schon öfter gesagt, muß es aber trotzdem immer wieder tun: Durch das Nachvollziehen fremder Programme kann man sehr viel lernen.

Inzwischen wissen Sie ja, daß alle Daten im Computer im Binärformat enthalten sind. Wie man eine normale, ganze Zahl zur binären umrechnet, soll Ihnen an einem einfachen Rechenbeispiel gezeigt werden. Als Beispiel nehmen wir

38. Der C 64 und Fließkommazahlen

die Zahl 1985. Man teilt diese Zahl so lange durch 2, bis das Ergebnis 0 wird. Jedesmal notiert man sich den Rest, der entweder 0 oder 1 sein kann:

1985	: 2=	992	Rest 1
992	: 2=	496	Rest 0
496	: 2=	248	Rest 0
248	: 2=	124	Rest 0
124	: 2=	62	Rest 0
62	: 2=	31	Rest 0
31	: 2=	15	Rest 1
15	: 2=	7	Rest 1
7	: 2=	3	Rest 1
3	: 2=	1	Rest 1
1	: 2=	0	Rest 1

Auch wenn Sie es noch nicht erkennen: Da steht schon das binäre Ergebnis. Von unten nach oben gelesen, ist das nämlich der Rest:

111 1100 0001

Nun reden wir ja von Fließkommazahlen. Also verändern wir unser Beispiel noch etwas. Jetzt soll uns die Zahl 1985,125 interessieren. In Kapitel 29 haben Sie gelernt, daß man das Komma verschieben kann, um daraus beispielsweise $1{,}985125 \times 10^3$ zu machen. Wir wollen uns das Verschieben des Kommas aber für etwas später aufheben und zunächst einmal außer dem schon umgewandelten Vorkommateil nun auch den Nachkommateil, also die »0,125«, ins Binärformat übertragen.

Genauso, wie wir vorhin eine Kettendivision durch 2 verwendet haben, gebrauchen wir nun eine Kettenmultiplikation mit 2. Der gesamte Nachkommateil wird dabei verdoppelt. Entweder ergibt sich dabei eine Vorkommastelle (das ist dann immer eine 1) oder das Ergebnis bleibt kleiner als 1. Wenn sich bei einem solchen Rechenschritt keine Vorkommastelle ergibt, schreibt man an die entsprechende Nachkommastelle der Binärzahl eine 0, andernfalls eine 1. Es wird solange verdoppelt, bis keine Nachkommastellen mehr zur Verfügung stehen. Das klingt ziemlich umständlich. Am besten sehen Sie sich das jetzt mal an unserem Beispiel an:

0,125 x 2 = 0,250   1. Nachkommastelle:0

Beim ersten Verdoppeln hat sich keine neue Vorkommastelle ergeben, deshalb ist die erste Nachkommastelle der Binärzahl eine Null.

0,25 x 2 = 0,5   2. Nachkommastelle:0

Auch beim zweiten Verdoppeln ermitteln wir keine neue Vorkommastelle, wodurch sich wieder eine Null als Nachkommastelle ergibt.

0,5 x 2 = 1,0   3. Nachkommastelle:1

Hier hat sich nun eine Vorkommastelle beim Verdoppeln gebildet: Daher taucht als 3. Nachkommastelle unserer Binärzahl eine 1 auf. Gleichzeitig war das die letzte Nachkommastelle, denn unsere Ausgangszahl weist nach dem Komma nun nur noch eine Null auf.

Zur Übung wollen wir noch eine andere Zahl mit Nachkommastellen ins Binärformat überführen, nämlich 0,1.

0,1x2 = 0,2   1. Nachkommastelle:0
0,2x2 = 0,4   2. Nachkommastelle:0
0,4x2 = 0,8   3. Nachkommastelle:0
0,8x2 = 1,6   4. Nachkommastelle:1

Jetzt läßt man - das habe ich beim ersten Beispiel noch nicht erwähnt - diese neue Vorkommastelle einfach weg und rechnet wieder mit den Nachkommastellen weiter:

0,6x2 = 1,2	5. Nachkommastelle:1
0,2x2 = 0,4	6. Nachkommastelle:0
0,4x2 = 0,8	7. Nachkommastelle:0
0,8x2 = 1,6	8. Nachkommastelle:1
0,6x2 = 1,2	9. Nachkommastelle:1

Das kommt Ihnen sicherlich von der 5. Verdoppelung her bekannt vor. Es zeigt sich, daß diese Rechnung nie aufgeht, weil sich eine periodische Zahl ergibt:

0,000 1100 1100 1100...

Das kann Ihnen öfters bei der Zahlenumwandlung passieren, daß ein endlicher Dezimalbruch in einen unendlichen periodischen Binärbruch übergeht. Kehren wir zurück zu unserem ersten Beispiel, 1985,125. Die ganze Umwandlung (Vorkomma- und Nachkommaanteil) führte zu:

111 1100 0001,001

Der dritte Schritt der Verwandlung von der Dezimalzahl zum Binärformat (nach 1.=Vorkommaanteil umwandeln, 2.=
Nachkommaanteil umwandeln) ist das sogenannte Normalisieren. Das ist einfach das Verschieben des Kommas nach links (wie in unserem Beispiel) oder rechts, solange, bis vor dem Komma nur noch Nullen stehen und direkt hinter ihm eine 1. In Kapitel 29 haben wir gelernt, daß für jede Stelle, die das Komma nach links wandert, der Exponent um 1 höher wird. Unser Exponent ist im Moment noch Null (2^0 ist ja 1). Um also nach der Regel zu normalisieren, wird das Komma um 11 Stellen nach links verschoben. Der Exponent ist dann 11(dez) und unsere Zahl erscheint im neuen Gewand:

0.1111 1000 0010 01 E +1011

E +1011 heißt dabei Exponent, und wird im Binärformat dargestellt (1011 (bin.) ≙ 11 (dez.)). Soweit, sogut. Alles bisher unternommene hat Allgemeingültigkeit. Von nun an aber müssen wir uns spezialisieren auf den Commodore 64 (in einigen anderen Computern ist es aber auch so). Der Exponent kann ja - je nachdem, ob das Komma nach links oder nach rechts zum Normalisieren verschoben wurde - positiv sein (wie bei unserem Beispiel) aber auch negativ. Im Commodore 64 wird zum Exponenten die Zahl 128 addiert. Das ist dann Schritt 4, der im Beispiel zu 139 führt, womit wir schon das Exponentenbyte fertig haben:

Exponent: dez.139 bin.1000 1011 hex.8B

Hätten wir einen negativen Exponenten erhalten, zum Beispiel 20, dann stünde im Exponentenbyte nun dez.108, beziehungsweise dasselbe im Binärformat.

Der Rest unserer Zahl, also die Mantisse, wird nun Schritt 5 unterzogen. Zunächst läßt man das Komma weg. Die Binärzahl wird dann auf 4 Byte linksbündig aufgeteilt. In unserem Beispiel erhalten wir so:

1111 1000	0010 0100	0000 0000	0000 0000
Byte 1	Byte 2	Byte 3	Byte 4

Wie Sie sehen, werden die unbenutzten Bits mit Nullen aufgefüllt. Was nun noch nicht berücksichtigt wurde, ist das Vorzeichen der Mantisse. Es ist im Beispiel noch nicht zu erkennen, ob wir +1985,125 oder -1985,125 vorliegen haben. Das gehen wir nun im letzten Schritt (Nummer 6) an. Im Commodore 64 gibt es zwei Möglichkeiten der Speicherung von Fließkommazahlen. Vor Schritt 6 muß man sich entscheiden, wo man die Zahl haben will.

In Kapitel 30 ist schon mal der FAC erwähnt worden, der Fließkomma-Akkumulator 1, welcher die Speicherstellen dez. 97 bis 102 ($61 bis $66) belegt. Ein zweiter Fließkomma-Akkumulator, AFAC oder ARG genannt, belegt die Plätze dez. 105 bis 110 ($69 bis $6E). Diese Akkumulatoren haben für die Fließkommarechnungen eine ähnliche Bedeutung wie der Akku für die 1-Byte-Rechnungen. Dort werden fast alle Ergebnisse abgelegt oder Zahlen abgerufen. Wir sehen, daß wir darin 6 Byte zur Verfügung haben. In Byte 97 liegt der Exponent in der von uns ermittelten Form. Byte 98 bis 101 sind die vier Mantissenbytes. Was ist in Byte 102? Das Vorzeichen! Bit 7 dieses Bytes ist 0, wenn eine positive, und 1 wenn eine negative Zahl vorliegt. Das galt für den FAC, wie Sie aus den Speicherstellen schon gesehen haben. Für den ARG ist das aber ganz genauso. Sehen wir uns nun in Bild 22 unsere Beispielzahl im FAC und im ARG noch mal an.

Im Bild ist auch angedeutet, daß die restlichen 7 Bit (Bits 0 bis 6) des Vorzeichen-Bytes keine Rolle spielen. Sie werden später direkt in diese Akkumulatoren hineinsehen und allerlei Bit-Müll darin finden. Lediglich Bit 7 ist für uns von Bedeutung.

Eigentlich ist das ja eine ganz schöne Verschwendung, von einem Byte wie diesem Vorzeichen-Byte lediglich ein einziges Bit zu nutzen. Wenn eine beliebige Fließkommazahl irgendwo im Computer abgespeichert wird, dann gilt ein anderes Format, das MFLPT-Format (von Memory-FLoating PoinT). Man speichert hier nur in 5 Byte. Das Vorzeichen-Byte fällt weg. Wie aber merkt sich der Computer das Vorzeichen? Das ist ganz schlau eingefädelt: Es gibt nämlich in den 5 Byte (1 Exponenten-Byte + 4 Mantissen-Byte) ein überflüssiges Bit. Sie werden sich sicher erstaunt fragen, wo?

FAC	$ / dez.	61 / 97	62 / 98	63 / 99	64 / 100	65 / 101	66 / 102
ARG	$ / dez.	69 / 105	6A / 106	6B / 107	6C / 108	6D / 109	6E / 110
Inhalt Binär		1000 1011	1111 1000	0010 0100	0000 0000	0000	
Hex.		8B	F8	24	00	0000	0... ...
Dez.		139	248	36	0	0	
Byte Nr.		1	2	3	4	5	6
Erläuterung		Exponent	Mantisse				Vorzeichen

Bild 22. So sieht die Zahl 1985, 125 komplett im FAC und ARG aus

Erinnern Sie sich doch bitte zurück an den Schritt 3, das Normalisieren. Dort wurde so verfahren, daß rechts vom Komma eine 1 steht. Wenn da aber immer und ganz grundsätzlich diese 1 steht, dann muß man sie sich eigentlich gar nicht mehr besonders merken. Man kann - vorausgesetzt, man berücksichtigt diese 1 im Bit 7 des ersten Mantissen-Bytes immer bei den Rechnungen - das Bit für andere Zwecke verwenden: Also als Vorzeichenbit. Taucht hier also eine 0 auf, dann liegt eine positive Zahl vor, ist es aber eine 1, dann signalisiert diese eine negative Zahl. Für das MFLPT-Format muß in unserem Beispiel also Bit 7 des ersten Mantissen-Bytes gelöscht werden (1985,125 ist ja nun mal positiv) und die komplette Zahl sieht im MFLPT-Format so aus:

1000 1011	0111 1000	0010 0100	0000 0000	0000 0000
Byte 1	↑Byte 2	Byte 3	Byte 4	Byte 5
Exponent	M A	N T	I S	S E

Der Pfeil weist auf das Vorzeichen-Bit. Man spricht hier auch vom »gepackten« Format. Damit das alles nun nicht nur graue Theorie bleibt und Sie auch aus eigenem Erleben diese Zahlenformate sehen können, wollen wir hier ein kleines Testprogramm ausprobieren. Es wird Ihnen auch später noch gute Dienste leisten können, wenn Sie mal irgendwelche Zahlen in das FLPT- (also FAC oder ARG) oder ins MFLPT-Format umrechnen müssen. »Zu Fuß« ist das ja

– wie Sie nun wissen – ganz schön haarig! Wie so oft, besteht auch dieses Programm aus einem Basic-Teil, der die Benutzerführung übernimmt und zwei kleinen Maschinenroutinen, die per USR-Vektor angesprungen werden. In diesen Assembler-Programmteilen sind zwei Interpreter-Routinen verborgen, die sehr nützlich und daher erklärenswert sind. Als Bild 23 ist das Basic-Aufrufprogramm abgedruckt.

Es fragt zunächst mal, ob der SMON geladen ist. Der wird nämlich aus dem Programm heraus angesprungen.

```
5 REM*** TEST FUER FLPT UND MFLPT ***          <162>
10 POKE 52,96:POKE 56,96:CLR:PRINT CHR$(14
   7)CHR$(17)CHR$(17)                          <215>
15 PRINT"IST DER SMON EINGELADEN?":INPUT"J
   /N";A$:IF A$="N"THEN END                    <254>
20 PRINT CHR$(17)CHR$(17)"{3SPACE}FLPT IN
   FAC"TAB(25)"1":PRINT                        <003>
30 PRINT"{3SPACE}FLPT IN ARG"TAB(25)"2":PR
   INT                                         <030>
40 PRINT"{3SPACE}MFLPT AB $6800"TAB(25)"3"
   :PRINT:PRINT                                <077>
50 GET A$:IF A$<"1"OR A$>"3"THEN 50            <211>
60 PRINT"AUSWAHL",A$:PRINT:PRINT:INPUT"GEB
   EN SIE EINE ZAHL EIN";Z                     <107>
65 PRINT CHR$(147)                             <142>
70 ON VAL(A$)GOTO 100,200,300                  <233>
100 REM***** FAC *****                         <097>
110 POKE 785,0:POKE 786,192:REM USR-VEKTOR
     AUF SMON = $C000                          <091>
120 A=USR(Z)                                   <205>
200 REM***** ARG *****                         <213>
210 POKE 785,0:POKE 786,96:REM USR-VEKTOR
    AUF $6000                                  <011>
220 A=USR(Z)                                   <049>
300 REM***** MFLPT *****                       <227>
310 POKE 785,0:POKE 786,97:REM USR-VEKTOR
    AUF $6100                                  <114>
320 A=USR(Z)                                   <150>
400 REM**********************************      <138>
410 REM NACH MELDUNG DES SMON MIT DEN          <042>
420 REM KOMMANDOS                              <221>
430 REM (1)   M 0061                           <212>
440 REM (2)   M 0069                           <231>
450 REM (3)   M 6800                           <241>
460 REM DEN MONITOR EINSCHALTEN. DIE           <137>
470 REM EINGEGEBENE ZAHL IST DANN ALS          <148>
480 REM HEX-BYTES SICHTBAR.                    <135>
490 REM**********************************      <228>

© 64'er
```

Bild 23. Testprogramm für die beiden kleinen Assembler-Routinen. Die Bedienung ist im Artikel erklärt.

Wird die Frage mit »J« beantwortet, dann zeigt sich ein kleines Menü, andernfalls ist das Programm beendet: Der SMON muß also erst geladen werden.

Das Menü bietet 3 Optionen: Eine Zahl kann im FAC (Option 1), im ARG (Option 2) oder im MFLPT-Format ab Speicherstelle $6800 (Option 3) betrachtet werden.

Für Option 1 wird der USR-Vektor auf die Einsprungadresse des SMON gestellt und dann mittels USR-Kommando die Zahl Z in den FAC übergeben. Es schaltet sich dann der SMON ein, der nun mittels des Kommandos M 0061 den Inhalt des FAC als Hex-Zahlen zeigt.

Option 2 richtet zunächst den USR-Vektor auf ein kleines Assembler-Programm ab $6000, welches den FAC-Inhalt in den ARG schiebt, dann den USR-Vektor auf den SMON richtet und schließlich auch diesen einschaltet. Auch hier wird mit dem M-Kommando dann per M 0069 der ARG-Inhalt sichtbar. Option 3 richtet den USR-Vektor auf eine Maschinenroutine, die bei $6100 beginnt. Dort wird der FAC-Inhalt nach $6800 und folgende Speicherstellen verschoben und zwar ins MFLPT-Format. Anschließend erfolgt dann wieder das Ausrichten des USR-Vektors auf den SMON, Anschalten des SMON, wo man durch M 68000C den Inhalt ansehen kann.

Folgende Vorgehensweise empfehle ich Ihnen:

1. Einladen des SMON

2. Eintippen der beiden kleinen Assembler-Routinen mit Hilfe des SMON und Speichern (mit dem SMON-Kommando S"Programmname", 08, 6000, 610A speichern).

2a. Wenn die beiden Routinen schon gespeichert vorliegen, dann laden Sie sie jetzt ein. Jedenfalls sollten Sie nach dem Laden beider Assembler-Programme (SMON und die beiden Routinen) NEW eingeben, so daß alle Zeiger zurückgestellt werden.

3. Erst jetzt Laden oder Eintippen des Basic-Aufrufprogrammes.

Wenn Sie nun das Testprogramm starten und zum Beispiel unsere Zahl 1985,125 eingeben, werden Sie folgendes finden:

```
Option 1:
M0061
:0061 8B F8 24 00 00 78 00 00

Option 2:
M0069
:0069 8B F8 24 00 00 78 D4 CE

Option 3:
M6800
:6800 8B 78 24 00 00 FF FF FF
```

Die Bytes, welche zu unserer Zahl gehören, sind unterstrichen. Sie können jeweils nach RUN/STOP noch mit dem SMON-Kommando $8B (oder eine andere Sie interessierende Hexzahl) eine Ausgabe im Binär- und im Dezimalformat erreichen.

So, nun aber endlich zu den beiden Assembler-Routinen. Zur Option 2 gehört das folgende, bei $6000 beginnende Programm:

```
6000   JSR   $BC0C
```

$BC0C ist die erste Interpreter-Routine, die wir uns zunutze machen. Sie schiebt den Inhalt vom FAC in den ARG. Mehr dazu später.

```
6003   LDA   #$00
6005   STA   $0311
6008   LDA   #$C0
600A   STA   $0312
```

Damit haben wir den USR-Vektor auf $C000 gestellt.

```
600D   JMP   $C000
```

Das war das Einschalten des SMON. Im Grunde genommen könnten wir uns das Stellen des USR-Vektors ersparen.

Es ist aber sinnvoll – vor allem bei langen Programmen – wenn verstellte Vektoren nach Beendigung des Programmes auf einem definierten Wert stehen.

Nun noch die Routine für Option 3:

```
6100   LDX   #$00
6102   LDY   #$68
6104   JSR   $BBD4
```

Auch das ist wieder eine Interpreter-Routine: Sie schiebt den FAC-Inhalt in einen Speicherbereich, dessen Startbyte durch die beiden Index-Register angegeben wird (X-Register für LSB, Y-Register für MSB, hier also $6800). Dabei wird die Zahl vom FLPT-Format in das MFLPT-Format umgewandelt. Das »Prögrämmchen« schließen wir ab mit einem Sprung zum Rest der ersten Routine:

```
6107   JMP   $6003
```

Sehen Sie sich mal einige Zahlen im Fließkomma-Format an. Fast alle Operationen mit Zahlen vollführt unser Computer mit diesen Fließkommazahlen. Das ist dann bei-

spielsweise der Grund dafür, daß aus einer Basic-Zeile wie der folgenden:

IF INT(X*10)=INT(ABS(X*10))THEN ...

auch bei positiven X-Werten (wo man mathematisch Gleichheit feststellt) manchmal die Bedingung als nicht erfüllt erkannt wird. X wird sofort als Fließkommazahl in den FAC gelegt, mit einer Fließkomma-Zehn multipliziert, der ABS-Wert wird ebenfalls per Fließkomma-Arithmetik ermittelt und so weiter. Dabei treten häufig Rundungsprobleme auf, wenn ein Zwischenergebnis mehr als 32 signifikante binäre Nachkommastellen aufweist (wie wir es ja zum Beispiel beim periodischen Binärbruch gesehen haben, der sich aus der simplen Dezimalzahl 0,1 ergibt). Das Rechnen mit Fließkommazahlen im Computer öffnet zwar einen ungeheuren Zahlenraum für unsere Anwendungen, es geht aber viel langsamer als die 2-Byte-Arithmetik. Immerhin müssen hier jedesmal 6 Byte (beziehungsweise 5 bei MFLPT) berücksichtigt werden. Ich glaube aber kaum, daß wir jemals in die Verlegenheit kommen werden, beispielsweise eine Fließkomma-Addition programmieren zu müssen. Eben weil unser C 64 fast alle Zahlenoperationen mit Fließkomma-Formaten durchführt, sind nahezu alle Eventualitäten schon als fertige abrufbare Programme im Interpreter enthalten. Wir müssen nur wissen, wie unsere Zahlen aussehen (das haben Sie nun ja gelernt) und wo und wie man sie für Operationen bereithält und wo und wie man die entsprechenden Routinen finden kann. Einen der wichtigsten Wege, unsere Zahlen ans Maschinenprogramm zu übergeben, haben Sie schon kennengelernt: Das Argument der USR-Funktion landet automatisch im FLPT-Format im FAC.

39. Die beiden ersten Interpreter-Routinen

Von nun an sollen nach und nach Interpreter-Routinen vorgestellt werden. Das ist allerdings nicht so einfach wie bei der Kernel-Sprungtabelle. Es gibt für die letzteren viele recht gut dokumentierte Listen. Für die Interpreter-Routinen ist kaum Literatur vorhanden. Will man die ähnlich erfassen wie die Kernel-Routinen, dann muß man ROM-Listings wälzen und vor allem probieren, probieren ... Falls Sie also mal einen Fehler in der Beschreibung feststellen oder Dinge, die ich leer lassen muß, weil mir dazu die Erleuchtung noch nicht gekommen ist, selbst schon kennen, dann schreiben Sie mir. Gemeinsam haben wir vielleicht die Chance, auch die letzte im Interpreter versteckte Nuß noch zu knacken!

Nun also zur ersten schon verwendeten Routine:

Name	MOVAF
Zweck	Übertragen des FAC in den ARG
Adresse	$BC0C, dez. 48140
Vorbereitung	Wert in FAC
Speicherstellen	$61-66 FAC, $69-6E ARG, $6F, $70
Register	Akku, X-Register
Stapelbedarf	4

Diese Routine ist deswegen so wichtig, weil viele Rechenoperationen, die zwei Zahlen verknüpfen, zwischen dem FAC und dem ARG abgewickelt werden. Wenn Sie unser kleines Testprogramm mal mit der Option 2 laufen lassen und hinterher nicht nur mit M0069 in den ARG, sondern auch mit M0061 in den FAC hineinsehen, dann stellen Sie fest, daß der FAC-Inhalt noch immer vorhanden ist.

Allerdings ist das nicht immer der Fall. MOVAF rundet nämlich – wenn nötig – vorher noch den FAC-Inhalt, der dann natürlich anders aussieht.

Fast noch häufiger benutzt man die zweite Interpreter-Routine:

Name	MOVMF
Zweck	Übertragung von FAC in Speicher unter Umrechnung ins MFLPT-Format
Adresse	$BBD4 dez. 48084
Vorbereitung	Wert in FAC, Zieladresse in X- und Y-Register (X = LSB, Y = MSB)
Speicherstellen	$61-66 FAC, $70, $22, $23
Register	Akku, X- und Y-Register
Stapelbedarf	4

Außer den unter »Speicherstellen« genannten sind natürlich auch noch die Zieladresse und deren vier nachfolgende Bytes in die Routine einbezogen (das MFLPT-Format besteht ja aus 5 Byte). $22/23 ist ein für die Operation verwendeter Zeiger.

MOVMF wird häufig dann verwendet, wenn Werte aus welchen Gründen auch immer, außerhalb der Fließkomma-Akkumulatoren gelagert werden müssen.

Es wird Ihnen vielleicht aufgefallen sein, daß im Gegensatz zur Beschreibung der Kernel-Routinen – die Rubrik »Fehler« fehlt. Der Grund ist, daß es keine solchen Sicherungen bei den Interpreter-Routinen gibt. Was passieren kann, ist unter bestimmten Bedingungen das Ansteuern von normalen Basic-Fehlermeldungen, die aber nicht immer den tatsächlichen Zustand wiedergeben. Wenn Ihnen mal bei der Programmierung mit Interpreter-Routinen Zweifel aufkommen, dann verfolgen Sie lieber den Programmweg mittels eines ROM-Listings und schalten Sie eigene Fehler-Routinen ein. Das war aber nur für die Fortgeschrittenen gesagt. Wir werden uns erst nach und nach dahin vortasten. Zunächst fehlen uns ja noch ein paar Assembler-Kenntnisse. Mit dem nächsten Abschnitt soll das noch besser werden.

40. Assembler-Befehle zum Beherrschen von Bits

Fangen wir also mit AND an. AND verknüpft den Akku-Inhalt Bit für Bit mit dem angegebenen Wert nach den Regeln der logischen UND-Verknüpfung. Die Adressiermöglichkeiten dieses Befehls sind allerlei:

AND 6000	absolut
AND FE	Zeropage absolut
AND #07	unmittelbar
AND 6000,X	absolut-X-indiziert
AND 6000,Y	absolut-Y-indiziert
AND (FA,X)	indiziert-indirekt
AND (FB),Y	indirekt-indiziert
AND FE,X	Zeropage-absolut-X-indiziert

Damit haben wir eine ganze Menge an Möglichkeiten. Erinnern Sie sich noch an die Regeln einer UND-Verknüpfung? Wenn nicht, dann sehen Sie sich noch mal die Tabelle 12 an.

Sie erkennen, daß zwei miteinander AND-verknüpfte Bits nur dann als Ergebnis 1 haben, wenn in beiden Bits der Wert 1 steht. Man kann mittels AND ganz gezielt Bits lö-

schen. Nehmen wir mal als Beispiel an, wir wollten geshiftete Zeichen (das sind die mit den Codes größer als 128) in normale Zeichen umwandeln. Dazu bringen wir die Zeichencodes in den Akku und löschen Bit 7. Übrig bleibt dann der Code für das ungeSHIFTete Zeichen. Für das Löschen von Bit 7 brauchen wir eine sogenannte UND-Maske, die dafür sorgt, daß alle anderen Bits unverändert bleiben. An den Stellen muß in dieser Maske also eine 1 stehen (denn 0 AND 1 ergibt 0, 1 AND 1 ergibt 1). Lediglich Bit 7 der Maske muß 0 sein. Die Maske muß also heißen:

0111 1111 $7F dez. 127

Nehmen wir an, im Akku befände sich der Code für ein geshiftetes A, also dez. 193 (binär 1100 0001, $C1), dann ergibt die AND-Verknüpfung mit der Maske:

```
Akku            1100  0001  Shift A
Maske           0111  1111
AND             ______________________
Jetzt im Akku   0100  0001
```

Normales A (Code dez. 65, $41)

Man kann also, je nach Wahl der Maske, beliebige Bits löschen.

AND ist, je nach der gewählten Adressierungsart, ein 2- oder 3-Byte-Befehl. Weil das Ergebnis im Akku steht, können Flaggen beeinflußt werden. Die N- und die Z-Flagge reagieren auf das Ergebnis.

Im Gegensatz zu Basic, wo es nur eine ODER-Verknüpfung gibt, nämlich OR, existieren im Assembler zwei davon. Man unterscheidet ein »inklusives« und ein »exklusives« ODER. Die inklusive ODER-Verknüpfung des Akku mit den angegebenen Daten geschieht mit dem Assembler-Befehl ORA. ORA entspricht dem Basic-Befehl OR. Alle Adressierungsarten, die dem AND-Befehl offenstehen, können auch auf ORA angewendet werden. Wenn man Bits ORA-verknüpft, findet man folgende Ergebnisse:

```
0 ORA   0 = 0
0 ORA   1 = 1
1 ORA   0 = 1
1 ORA   1 = 1
```

AND	0	1
0	0	0
1	0	1

Tabelle 12. Wahrheitstabelle zur AND-Verknüpfung

ORA	0	1
0	0	1
1	1	1

Tabelle 13. Wahrheitstabelle zur ORA-Verknüpfung

EOR	0	1
0	0	1
1	1	0

Tabelle 14. Wahrheitstabelle zur EOR-Verknüpfung

Auch hier ist eine sogenannte Wahrheitstabelle recht einprägsam (siehe Tabelle 13).

Während man mit AND gezielt Bits löschen kann, ist es mit ORA möglich, Bits zu setzen. Auch dazu verwendet man eine Maske, die an allen Stellen, an denen Bits unverändert bleiben sollen, eine 0, sonst aber eine 1 enthält. Nehmen wir nochmal das Beispiel von vorhin und wandeln nun das ungeSHIFTete Zeichen in ein geSHIFTetes um. Wir müssen also Bit 7 wieder setzen: Da muß in der Maske dann eine 1 stehen. Alle anderen Bits bleiben unverändert, wenn die Maske dort eine Null aufweist. Die Maske muß daher heißen:

1000 0000 $80 dez. 128

Im Akku soll das ungeSHIFTete B stehen (Code dez. 66, $42, bin. 0100 0010). Die Rechnung sieht dann so aus:

```
Akku            0100  0010  Code für B
Maske           1000  0000
  ORA           ______________________
Jetzt im Akku   1100  0010
```

Code für geshiftetes B.

Je nach Art der Maske kann man also ein oder mehrere Bits setzen. Im Beispiel ist auch der Einfluß dieses Befehls auf die Flaggen zu erkennen. Der Akku-Inhalt vor der ORA-Operation hatte kein Bit 7, also keine gesetzte N-Flagge. Danach ist Bit 7 gesetzt und die N-Flagge zeigt eine 1. Außer der N-Flagge kann – ebenso wie beim AND-Befehl – auch noch die Z-Flagge reagieren. ORA ist je nach Adressierungsart ein 2- oder 3-Byte-Befehl.

Während zwei Bit in der ORA-Verknüpfung eine 1 ergeben, wenn sie beide gesetzt sind oder eines von beiden, schließt die EOR-Verknüpfung den ersten Fall aus. EOR ist die exklusive ODER-Verknüpfung. Sie läßt sich sprachlich erfassen im »entweder ... oder ...«, also beispielsweise: Beim Roulette fällt die Kugel entweder auf Rouge oder auf Noir, beides zusammen ist nicht möglich. Die Regeln bei EOR sind also:

```
0   EOR 0 = 0
0   EOR 1 = 1
1   EOR 0 = 1
1   EOR 1 = 0
```

Eine Wahrheitstabelle dazu sehen Sie in Tabelle 14.

Wozu verwendet man EOR? Es fällt Ihnen vielleicht auf, daß wir die aus Basic bekannte NOT-Funktion nicht in Assembler vorliegen haben. Obwohl EOR einige viel weitergehendere Verwendungsmöglichkeiten aufweist als NOT (aber auf Boolesche Algebra wollen wir hier nicht eingehen), kann man es mit gleicher Wirkung einsetzen. Wir haben beispielsweise in den ersten Kapiteln negative Zahlen durch Komplementieren erzeugt. Dabei sollte jedes Bit in sein Gegenteil verkehrt werden. Das wäre die Aufgabe einer NOT-Funktion. Durch ein EOR FF können wir dasselbe erreichen. Sehen wir uns wieder ein Beispiel an. Im Akku steht dez. 15 ($0F, bin. 0000 1111):

```
Akku            0000  1111
Maske           1111  1111   =$FF
  EOR           ______________________
Jetzt im Akku   1111  0000
```

Einerkomplement von dez. 15.

Auch EOR kann alle Adressierungsarten verkraften, die die beiden anderen logischen Assembler-Befehle erlauben. Je nach der gewählten Art liegt dann ein 2- oder 3-Byte-Befehl vor. Auch hier werden die Z- und die N-Flagge beeinflußt.

Das waren also die logischen Befehle. Leider ist hier nicht der geeignete Ort, die Vielseitigkeit, die damit möglich ist, deutlich zu machen. Wenn Sie sich dafür interessieren, sollten Sie mal etwas über Boolesche Algebra lesen oder eine Einführung in die mathematische Logik.

Um dieses Thema abzuschließen, soll noch erwähnt werden, daß der Basic-Interpreter so eingerichtet ist, daß er immer dann, wenn die Richtigkeit einer Aussage zu überprüfen ist, mit »–1« antwortet bei wahrer Aussage, dagegen mit »0« bei falscher. Auf diese Weise kommen diese merkwürdigen Basic-Programmzeilen ins rechte Licht, in denen Sequenzen auftauchen wie:

C=A-161-33*(A< 255)-64*(A < 192)-32*(A <160)+32*(A < 96)-64*(A < 64).

Jedesmal, wenn zum Beispiel A < 64 ist, tritt anstelle der Klammer ein »–1« auf. Übrigens ist diese Formel eine schöne kurze Möglichkeit, ASCII-Code (hier A als Variable) in

den Bildschirmcode umzurechnen (der Bildschirmcode steht dann in der Variablen C).

Kommen wir nun zur zweiten Gruppe von Assembler-Befehlen, die Bit-Manipulationen erlauben: den Verschiebe-Befehlen. Fangen wir dabei mit ASL an, was vom englischen »Arithmetic Shift Left« kommt. Zu deutsch heißt das dann »arithmetisches nach links schieben«. Davon sind wir aber auch noch nicht schlauer. Sehen wir uns an, was dieser Befehl tut (Bild 24).

Daraus folgt, daß immer dann, wenn man sich nicht hundertprozentig sicher ist, eine Abfrage des Carry-Bits erfolgen sollte, sofern man ASL zum Rechnen einsetzt (BCC beziehungsweise BCS bieten sich da an). Dazu kommen wir noch. Sehen wir uns zunächst mal an, wie ASL adressierbar ist:

ASL

ohne Adresse, der Akkuinhalt wird nach links verschoben. Manchmal als eigene Adressierungsart bezeichnet.

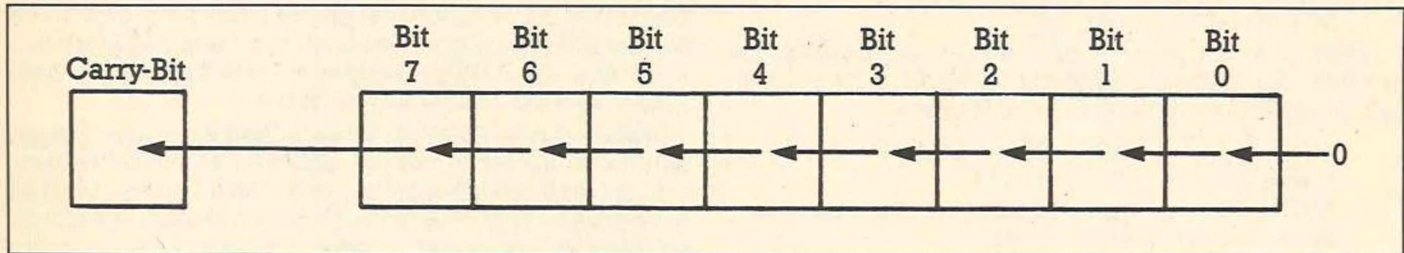


Bild 24. Wirkung des ASL-Befehls: Arithmetisches Linksschieben

Der gesamte Inhalt des Akku beziehungsweise der Speicherstelle (je nach Adressierung) wird um eine Bit-Position nach links verschoben. Das vorherige Bit 7 wandert in die Carry-Flagge, alle anderen Bits erhalten eine um 1 höhere Position, das freigewordene Bit 0 wird mit einer 0 aufgefüllt. Toll! Aber was soll das? Zur Erklärung machen wir noch mal einen kurzen Ausflug zu unserem normalen dezimalen Zahlensystem. Nehmen wir mal die Zahl 123. Bei der Einführung in die Fließkommazahlen haben wir gelernt das Komma zu verschieben. 123 ist ja dasselbe wie 123,00. Wenn wir das Komma um eine Stelle nach rechts verschieben, erhalten wir 1230,0 (dabei lassen wir jetzt mal den Exponenten außer acht, der wäre ja –1, weil 123,00 = $1230{,}0 \times 10^{-1}$). Man kann das Ganze auch andersherum sehen: Wir haben die Zahl 123 eine Stelle nach links verschoben und die freigewordene Stelle ganz rechts mit einer Null aufgefüllt. 1230,0 ist das Zehnfache von 123,00. Die Verschiebung um eine Stelle nach links hat also zur Multiplikation unserer Zahl mit der Basis unseres Zahlensystems (also 10) geführt. Eine zweimalige Linksverschiebung führt zu 12300, den 100fachen Wert unserer Ausgangszahl. Wir haben also die Zahl 123,00 x 10 x 10 genommen, das sind 10^2. Jede Linksverschiebung erhöht unseren Ausgangswert um eine Zehnerpotenz, oder – anders ausgedrückt – erhöht den Multiplikator um eine Zehnerpotenz und deshalb natürlich auch das Ergebnis (einmal linksschieben: Multiplikator = 10 = 10^1, zweimal linksschieben: Multiplikator: 100 = 10^2 und so weiter).

Im Binärsystem, zu dem wir nun wieder zurückkehren, ist die Zahlenbasis die Zahl 2. Einmal Linksschieben entspricht dann einer Multiplikation mit $2^1 = 2$. Das zweimalige Linksschieben führt zur Multiplikation mit $2^2 = 4$ und so weiter. Nehmen wir als Beispiel die Zahl 3, welche im Binärsystem 0000 0011 heißt:

1. ASL	führt zu	0000 0110	= dez.6 (2^1x3=2x3=6)
2. ASL		0000 1100	= dez. 12 (2x6= 12, 2^2x3=4x3=12)
3. ASL		0001 1000	= dez.24 (2x12=24, 2^3x3=8x3=24)
4. ASL		0011 0000	= dez. 48 (2x24=48, 2^4x3=16x3=48)
5. ASL		0110 0000	= dez. 96 (2x48=96, 2^5x3=32x3=96)
6. ASL		1100 0000	= dez. 192 (2x96=192, 2^6x3=63x3+192)

Bis jetzt landete im Carry-Bit immer eine Null. Wenn wir nun noch mal linksschieben, finden wir darin eine 1, die offensichtlich als Bit 8 unseres Ergebnisses dienen muß:

7. ASL (1) 1000 0000 = (mit Carry als Bit 8) dez. 384 (2x192=384, 2^7x3= 128x3=384

```
ASL 6000     absolut
ASL FE       Zeropage-absolut
ASL 6000,X   absolut-X-indiziert
ASL FA,X     Zeropage-absolut-X-indiziert
```

Je nach Adressierung tritt ASL dann als 1-, 2- oder 3-Byte-Befehl auf. Die N-, die Z- und die Carry-Flagge werden beeinflußt. Das Ergebnis steht bei der ersten Adressierungsart (also ASL ohne Adresse) im Akku. In den anderen Fällen findet man es in der jeweiligen Speicherstelle.

Nun gut, werden Sie sagen, man kann also mittels ASL Zahlen mit 2, 4, 8, 16 32 etc. multiplizieren. Was aber, wenn man mal 40 nehmen will? Da gibt es einige Möglichkeiten, die ein bißchen den Erfindungsgeist ansprechen. Man kann ja, wenn irgendeine Zahl Z mal 40 gerechnet werden soll, dafür schreiben:

40*Z=(32+8)*Z=32*Z+8*Z

Schon haben wir wieder Multiplikatoren, die den Einsatz von ASL ermöglichen. Die beiden Zwischenergebnisse (als 32xZ und 8xZ) speichern wir irgendwo ab und zählen sie dann zusammen. Wenn Z zum Beispiel 3 wäre, könnte man das so programmieren:

```
6000   STA   $6100
```

Dabei sollte im Akku Z also die 3 stehen, die wir nun zwischengespeichert haben.

```
6003   ASL
6004   ASL
6005   ASL
6006   ASL
6007   ASL
```

Jetzt liegt im Akku der 32fache Wert von 3, also 96 vor und wir speichern dieses Zwischenergebnis ab.

```
6008   STA   $6101
600B   LDA   $6100
```

Wir haben nun den Wert 3 aus dem Zwischenspeicher $6100 wieder in den Akku geholt und schieben ihn 3mal nach links, um den 8fachen Wert zu erhalten.

```
600E   ASL
600F   ASL
6010   ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8xZ noch im Akku.

```
6011   CLC
6012   ADC   $6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiter verwendet werden.

Auf diese Weise kann man immer einen Multiplikator in eine Zweierpotenz (2, 4, 8, 16,...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am besten mit BCC verzweigt, wenn die 8 Bit des Ergebnisses überlaufen.

manche Berechnungen von Bedeutung ist, muß das Carry-Bit irgendwie erfaßt werden. Wie man das erreicht, lernen wir noch. Leider ist diese Art der Division mittels LSR nicht so einfach verwendbar wie die Multiplikation mittels ASL. Während man dort durch geschicktes Aufteilen des Faktors ASL auch bei anderen Multiplikatoren als reine Zweierpotenzen anwenden konnte, ist das hier nicht so ohne weiteres möglich. Bei Divisionen geht man deshalb lieber andere Wege. Die zeige ich Ihnen ebenfalls etwas später.

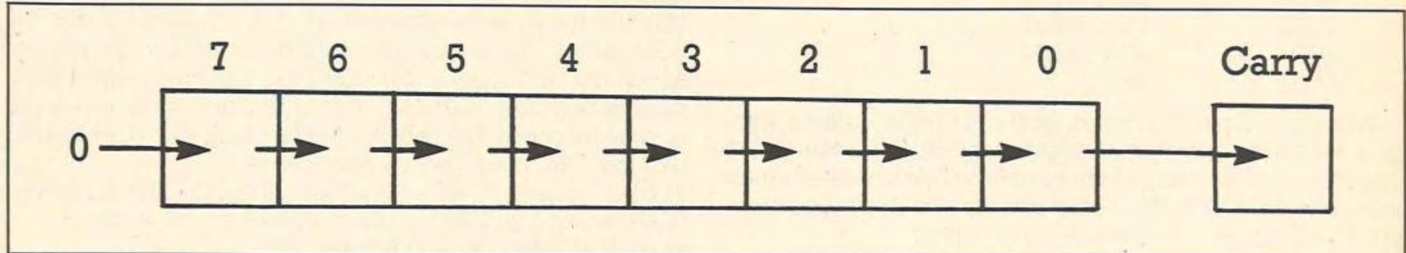


Bild 25. Wirkung von LSR auf ein Byte

Abschließend finden Sie in Tabelle 15 noch alles Wissenswerte zu den neuen Befehlen.

41. Die restlichen Bit-Verschiebe-Operationen

Da wäre zunächst einmal das Gegenstück zu ASL. Dort ging es ja um das nach-links-schieben. Jetzt schieben wir nach rechts. LSR heißt der dazu nötige Befehl. Das kommt von »logical shift right« und heißt zu deutsch »logisches Rechtsschieben«. Fragen Sie mich bitte nicht, weshalb »logisches«. Jedenfalls ist LSR ebenso für logische Bitprüfungen geeignet wie ASL.

Mittels LSR wird jedes Bit der adressierten Speicherstelle um einen Platz nach rechts geschoben. An die Stelle des Bit 7 tritt eine Null und Bit 0 wandert in das Carry-Bit (siehe Bild 25).

Erinnern Sie sich noch an das dezimale Linksschieben mit ASL? Wir hatten festgestellt, daß jedes Linksschieben einer Dezimalzahl einer Multiplikation mit 10 entspricht. Hier im umgekehrten Fall, also beim Rechtsschieben, muß jedes LSR einer Division durch 10 entsprechen:

25000	wird durch LSR	2500
2500	zu "	250
250	"	25

und so weiter

Geht man von der Ausgangszahl (25000) aus, dann ergibt sich der erste rechts verschobene Wert durch Division mit

$10^1=10$
der 2. durch $10^2=100$
der 3. durch $10^3=1000$, etc.

Es wird also durch Potenzen der Zahlenbasis 10 geteilt. Haben wir es – wie im Computer – mit Binärzahlen zu tun, deren Basis die 2 ist, dann teilen wir mit jedem LSR durch 2. Je nachdem, wie oft hintereinander das LSR auf eine Zahl ausgeübt wird, teilt man dann durch $2^1=2$, $2^2=4$, $2^3=8$, etc. Das konnte man sich alles ja schon vorstellen, nachdem ASL zur Multiplikation verwendet wurde. Auch hier muß man immer das Carry-Bit abfragen, denn die Division kann ja unter Umständen nicht aufgehen, wie das folgende Beispiel der Division von 3 durch 2 zeigt:

(3) 0000 0011 ergibt durch LSR: 0000 0001 und 1 im Carry-Bit. Das Ergebnis ist schon richtig, nämlich 1. Im Carry steht der Rest dieser Division, die 1. Weil der Rest für

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4*	N, Z
	abs.-Y-indiz.	3	39	57	4*	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5*	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4*	N, Z
	abs.-Y-indiz.	3	19	25	4*	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5*	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4*	N, Z
	abs.-Y-indiz.	3	59	89	4*	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5*	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

* bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 15. Alles Wissenswerte über die neuen Assembler-Befehle

LSR kann auf die gleiche Weise adressiert werden wie ASL:

LSR	auf den Akku bezogen
LSR 6000	absolut
LSR FE	Zeropage-absolut
LSR 6000,X	absolut-X-indiziert
LSR FA,X	Zeropage-absolut-X-indiziert

Im ersten Fall steht das Ergebnis im Akku, in den anderen Fällen in der jeweils adressierten Speicherstelle. Außer der N-Flagge, die in jedem Fall 0 wird, beeinflußt LSR auch die Carry-Flagge und unter Umständen die Z-Flagge. Je nach

Adressierungsart liegt LSR als 1-Byte-, 2-Byte- oder 3-Byte-Befehl vor.

Sowohl bei ASL als auch bei LSR hatten wir festgestellt, daß man herausgeschobene Bits, falls sie noch von Bedeutung sind, irgendwie aus dem Carry-Bit (dort sind sie ja gelandet) an einen sinnvollen Ort schaffen muß. Das ist natürlich möglich über eine Befehlskette, in der zunächst das Carry-Bit abgefragt wird:

zum Beispiel:

```
6000    BCC $6007
6002    LDA #$01
6004    STA $8000
6007    etc.
```

Wenn das Carry-Bit frei ist, wird alles weitere übersprungen. Wenn da drin etwas aufgetaucht ist, lädt man eine 1 (die ist ja im Carry-Bit) an die benötigte Speicherstelle (hier zum Beispiel 8000). Das kostet aber einige Bytes Speicherplatz und einige Taktzeiten Rechendauer.

Außerdem erschwert sich die Programmierung, wenn man eine Zahl öfter verschiebt und dann nach $8000 alle Carry-Inhalte packen will. So kompliziert brauchen wir auch gar nicht zu arbeiten, denn unsere CPU kennt zwei Befehle, die das Bit-Verschieben und das Carry-Verschieben für uns übernehmen.

Das sind:

ROL	rotate left	Linksrotieren
ROR	rotate right	Rechtsrotieren

Sehen wir uns zunächst mal ROL (Bild 26) an:

Wie bei ASL wandert jedes Bit um eine Position nach links. Das Bit 7 wird dabei in das Carry-Bit verschoben. In Bit 0 gelangt aber hier nicht eine 0 (wie bei ASL), sondern

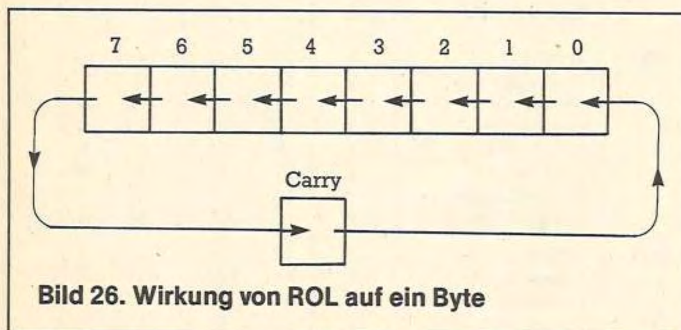


Bild 26. Wirkung von ROL auf ein Byte

der Inhalt des Carry-Bit (wohlgemerkt der Inhalt, der dort war, bevor dort Bit 7 hineingeschoben wurde). Bevor wir auf den praktischen Nährwert dieses Befehls eingehen, sollen erstmal die Adressierungsmöglichkeiten aufgeführt werden:

ROL		auf den Akku bezogen
ROL	6000	absolut
ROL	FE	Zeropageabsolut
ROL	6000,X	absolut-X-indiziert
ROL	FE,X	Zeropage-absolut-X-indiziert

Je nach Adressierung kann es sich dann wieder um einen 1-Byte- bis 3-Byte-Befehl handeln. Die N-, Z- und natürlich die Carry-Flagge sind beeinflußt und das Ergebnis des Befehls ist im Akku zu finden (erste Adressierungsart) oder in der angesprochenen Speicherstelle.

Wozu das Ganze? Abgesehen von der Möglichkeit, einzelne Bits auf diese Weise ohne Verlust aus einem Byte durch das Carry-Bit herausschieben zu können, um sie Prüfungen zu unterziehen, gibt es noch die Möglichkeit, einen Überlauf bei Rechenoperationen aufzufangen. Erinnern Sie sich an Kapitel 40, in dem wir mittels ASL Multiplikationen durchgeführt haben? Dort kann es unter gewissen Umständen ja leicht geschehen, daß ein Byte für das Ergebnis nicht mehr ausreicht. Wir haben in den Beispielen schon die Überlegung durchgeführt, daß man mittels BCC oder BCS prüfen sollte, ob man eine signifikante Stelle (also eine führende 1) aus dem Byte herausgeschoben hat. Ist das der Fall, dann gibt es zwei Wege:

1) Man veranlaßt den Ausdruck eines OVERFLOW ERROR, wenn nur 1-Byte-Zahlen zulässig sind, oder
2) man schaltet um auf 2-Byte-Zahlen.

Sehen wir uns das mal an dem Schritt 7 des Beispiels aus Kapitel 40 an. Dort hatten wir die Zahl 192 (binär 1100 0000) vorliegen (zum Beispiel in Speicherstelle 7000). Im Computer werden 2-Byte-Integers in der Form LSB/MSB verarbeitet. Wir schaffen also die Speicherstelle für das MSB von 192 in 7001. Jetzt muß dort noch 0 drin stehen. Um bei nochmaliger Multiplikation mit 2 eine 16-Bit-Zahl als Ergebnis zu erhalten, verfährt man wie folgt:

6000	ASL $7000	Damit ist die führende 1 ins Carry-Bit gewandert
6003	BCC $6008	Das setzt man natürlich nur dann ein, wenn man nicht genau weiß, welches Ergebnis zu erwarten ist. Wenn keine 1 ins Carry-Bit gelangte, kann man die nächste Zeile überspringen.
6005	ROL $7001	Damit wurde der Inhalt des Carry-Bit als Bit 0 ins MSB unseres Ergebnisses geschoben.
6008	etc.	

Die Funktion dieser Befehlssequenz können Sie aus Bild 27 entnehmen.

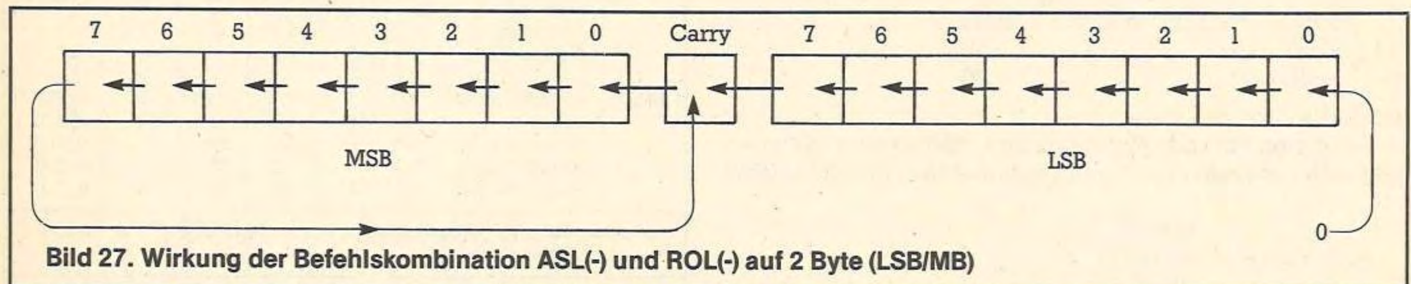


Bild 27. Wirkung der Befehlskombination ASL(-) und ROL(-) auf 2 Byte (LSB/MB)

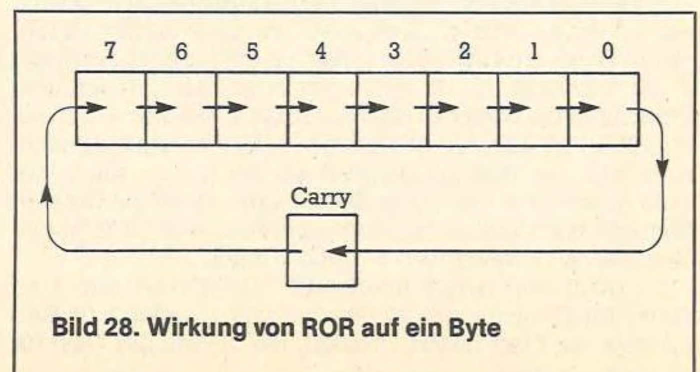


Bild 28. Wirkung von ROR auf ein Byte

Diesem Befehl werden wir später bei der 16-Bit-Multiplikation und Division noch häufig begegnen.

Sehen wir uns nun noch den letzten der Bit-Verschiebebefehle an: ROR. In Bild 28 ist schematisch gezeigt, wie rotiert wird.

Jedes Bit wandert, wie bei LSR, um eine Stelle nach rechts. Als Bit 7 kommt (im Gegensatz zu LSR) der Inhalt des Carry-Bit herein. Bit 0 wird ins Carry-Bit geschoben. Adressiert werden kann ROR ebenso wie ROL:

ROR		auf den Akku bezogen
ROR	6000	absolut
ROR	FE	Zeropage-absolut
ROR	6000,X	absolut-X-indiziert
ROR	FE,X	Zeropage-absolut-X-indiziert

Auch für die Byteanzahl, den Ort des Ergebnisses und die Flaggenbeanspruchung gilt dasselbe wie für ROL.

Die Einsatzmöglichkeiten für ROR sind allerdings geringer. Bei 16-Bit-Divisionen kann man zwar ROR einsetzen, um einen Unterlauf des MSB ins LSB aufzufangen. Weil man aber meist ohnehin andere Divisionsverfahren verwendet als das oben gezeigte mit LSR, erübrigt sich diese Anwendung in den meisten Fällen. Gut kann man ROR zu Bitprüfungen einsetzen. Das soll im nächsten Abschnitt an einem kleinen Beispiel gezeigt werden.

Zuvor aber noch eine Bemerkung: Wir sind nun durch den Befehlssatz des 6502-Assemblers fast hindurchgedrungen. Es fehlen uns nur noch - wenn ich mich nicht versehen habe - vier Befehle. Die allerdings hängen eng mit dem sogenannten Interrupthandling zusammen, das uns wohl einige Zeit beschäftigen wird.

42. Schneller Joystick

Vor einiger Zeit (64'er, Ausgabe 2/85) veröffentlichte P. Siepen eine Routine zur Abfrage des Joystickports, die eine interessante Leserbrief-Reaktion hervorrief. M. Hartig sandte nämlich einen Verbesserungsvorschlag, in dem der uns interessierende Befehl ROR die Hauptrolle spielt. Bevor ich die allerdings vorstelle, muß erst noch geklärt werden, was und wie abgefragt wird.

Signale vom Joystick landen in den DATA-Ports A oder B des CIA 1. CIA heißt »Complex Interface Adapter« und ist die Institution unseres Computers, die den Verkehr mit der Außenwelt erlaubt. Wir haben zwei Stück davon (CIA 1 und CIA 2). Je nachdem, in welchen Port der Joystick gesteckt wurde, laufen die Signale in den Registern $DC00 oder $DC01 (dezimal 56320 oder 56321) ein. Wir nehmen im weiteren mal $DC00 an. Die Bits 0 bis 4 beziehen sich auf den Joystick:

Bit 0	oben
Bit 1	unten
Bit 2	links
Bit 3	rechts
Bit 4	Feuerknopf

Wenn keine dieser Möglichkeiten angesprochen ist, enthalten diese Bits den Wert 1. Drückt man beispielsweise den Feuerknopf, dann wechselt der Inhalt von Bit 4 zum Wert 0. Man muß also ständig diese Bits überprüfen und reagieren, sobald eines davon 0 wird. Die Lösung von P. Siepen, diese Abfrage in das Interruptprogramm einzubauen, ist sehr brauchbar. Dadurch hat der Computer die Möglichkeit, trotzdem an anderen Aufgaben weiterzuarbeiten. Wir werden in den nächsten Folgen auf diese Programmiertechnik eingehen. Die Verbesserung von M. Hartig besteht

Name	UMULT
Zweck	Multiplikation zweier 16-Bit-Zahlen
Adresse	$B357 dez. 45911
Vorbereitungen	Faktor 1 in $28/29 Faktor 2 in $71/72
Speicherstellen	$28/29,$71/72, $5D
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 29. Die Interpreterroutine UMULT

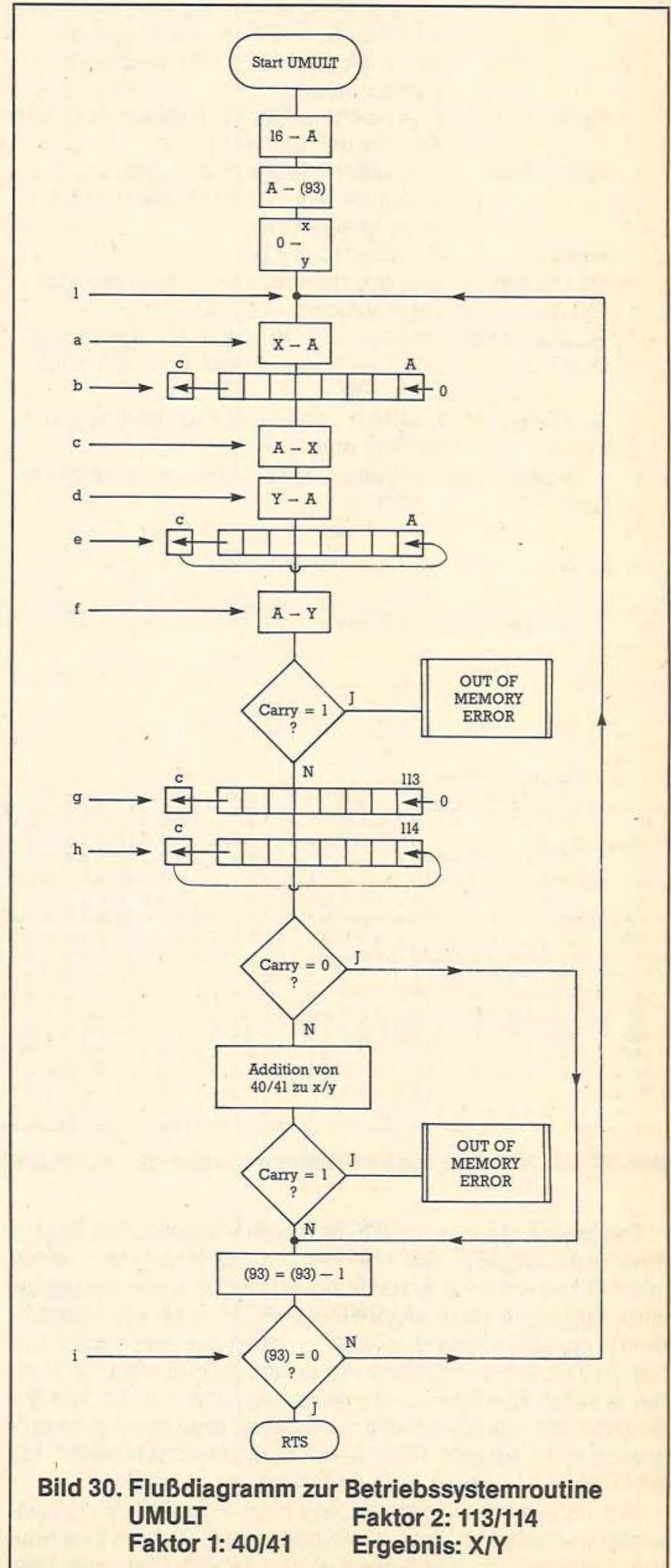


Bild 30. Flußdiagramm zur Betriebssystemroutine UMULT Faktor 1: 40/41 Faktor 2: 113/114 Ergebnis: X/Y

darin, daß er nicht durch CMP-Befehle den Inhalt von $DC00 prüft (was Zeit und auch Speicherplatz kostet), sondern mittels ROR Bit für Bit nach rechts in das Carry-Bit schiebt und dieses dann mit BCC abfragt. Sobald die Carry-Flagge nämlich frei ist, ist die zu dem Bit gehörige Joystick-Funktion gefragt.

Nun die Abfrageroutine:

```
LDA $DC00   Inhalt des DATA-Port A in den Akku
ROR         Durch Rechts-Rotieren wird Bit 0 in
            die Carry-Flagge geschoben.
BCC Oben    Wenn die Carry-Flagge nicht gesetzt
            ist, war Bit 0 eine Null, also die Joy-
            stickfunktion »Oben« gefordert, zu
            deren Bearbeitung hier verzweigt
            werden kann.
ROR         Das nächste Rechts-Rotieren schiebt
            Bit 1 in die Carry-Flagge.
BCC Unten   Auch hier wieder Abzweigen zur Be-
            arbeitung von »Unten«, wenn Bit 1
            nicht gesetzt war.
ROR         Bit 2 ins Carry-Bit
BCC Links   und bearbeiten, wenn nicht gesetzt
ROR         Bit 3 in Carry-Flagge
BCC Rechts  und verzweigen wenn Bit 3 Null war
ROR         zu guter Letzt kommt noch Bit 4 ins
            Carry-Bit
BCC Fire    und kann bearbeitet werden, wenn
            es Null war.
...weitere Bearbeitung, wenn keine Joystick-Funk-
tion
```

Gedanken machen muß, wie man sie am besten multipliziert. Was fehlt, ist ein allgemein gültiges Programm, das in der Lage ist, jede Zahlenkombination (solange es sich um 2-Byte-Integers handelt und das Ergebnis als 16-Bit-

43. Die 16-Bit-Manipulation

Zahl darstellbar ist) zu verarbeiten. Und da haben wir mal wieder Glück: Gut versteckt befindet sich so etwas bereits fertig in unserem Computer. Ab dez. 45900 ($B34C) liegt im Interpreter solch eine Routine und ihr Einsprungspunkt ist für uns bei dez. 45911 ($B357). Bevor wir aber detailliert darauf eingehen, soll noch das Prinzip erklärt werden, das dabei genutzt wird.

Jeden Tag rechnen Sie wahrscheinlich völlig automatisch Multiplikationsaufgaben, ohne noch Gedanken daran zu verschwenden, wieviel Schweiß das Erlernen dieser Technik früher mal gekostet hat. Könnten Sie heute noch jemandem genau erklären, warum man da was wie macht? Genau das müssen wir aber tun, damit der Binärautomat (unser C 64) multiplizieren lernt. Nehmen wir mal eine Multiplikation von 16 x 15:

```
16 x 15
 16
  80
 240
```

Daß wir nicht so genau wissen, was wir da tun, liegt am ziemlich komplizierten Zehnersystem. Damit das alles ein-

	114	113	Y	X	A	93	C	
I	00000000	01000001	00000000	00000000	00010000	00010000	–	Ausgangslage
a				–	00000000		–	
b					00000000		0	
c				00000000	–		0	
d				——→	00000000		0	
e					00000000		0	
f			00000000	←——			0	
g		10000010					0	
h	00000000						0	Ende 1.
i	00000000	10000010	00000000	00000000	00000000	00001111	0	Schleife
II	00000001	00000100	00000000	00000000	00000000	00001110	1, 0	Ende 2. Schleife
III	00000010	00001000	00000000	00000000	00000000	00001101	0	Ende 3. Schleife
IV	00000100	00010000	00000000	00000000	00000000	00001100	0	Ende 4. Schleife
V	00001000	00100000	00000000	00000000	00000000	00001011	0	Ende 5. Schleife
VI	00010000	01000000	00000000	00000000	00000000	00001010	0	Ende 6. Schleife
VII	00100000	10000000	00000000	00000000	00000000	00001001	0	Ende 7. Schleife
VIII	01000001	00000000	00000000	00000000	00000000	00001000	1, 0	Ende 8. Schleife
IX	10000010	00000000	00000000	00000000	00000000	00000111	0	Ende 9. Schleife
X	00000100	00000000	00000001	01000001	00000001	00000110	1, 0	Ende 10. Schleife
XI	00001000	00000000	00000010	10000010	00000010	00000101	0	Ende 11. Schleife
XII	00010000	00000000	00000101	00000100	00000101	00000100	1, 0	Ende 12. Schleife
XIII	00100000	00000000	00001010	00001000	00001010	00000011	0	Ende 13. Schleife
XIV	01000000	00000000	00010100	00010000	00010100	00000010	0	Ende 14. Schleife
XV	10000000	00000000	00101000	00100000	00101000	00000001	0	Ende 15. Schleife
XVIa,		00000000	01010000	01000000	01010000		0	
b,	00000000	00000000	01010001	10000001	01010001	00000000	1, 0	Ende 16. Schleife

Bild 31. UMULT am Beispiel der Multiplikation 321 x 65=20865

Der Vorteil dieser nur 18 Byte langen Unterroutine liegt in ihrer Schnelligkeit: Sie braucht nur 24 Taktzyklen, wenn nicht verzweigt wird, beziehungsweise 25, wenn verzweigt wird. Natürlich wäre anstelle von ROR auch die Verwendung von LSR möglich gewesen, denn die herausgeschobenen Bits werden nicht mehr benötigt. Will man nach einer solchen Abfrage wieder den Ausgangszustand des Akku oder der Speicherstelle herstellen, muß man eine entsprechende Anzahl ROR-Anweisungen anschließen, bis Bit 0 wieder in seine Ausgangslage rotiert ist.

Wir haben bisher gelernt, wie man 8-Bit-Zahlen miteinander malnehmen kann um 8- oder 16-Bit-Zahlen zu erhalten. Dabei ist unbefriedigend, daß man sich über jede Zahl

facher und überschaubarer wird, wechseln wir mal ins Binärsystem: 16 = 10000, 15 = 1111.

Die Aufgabe sieht dann so aus:

```
10000 * 1111
10000
 10000
  10000
   10000
11110000
```

Jetzt wird schon deutlicher, was wir getan haben. Der

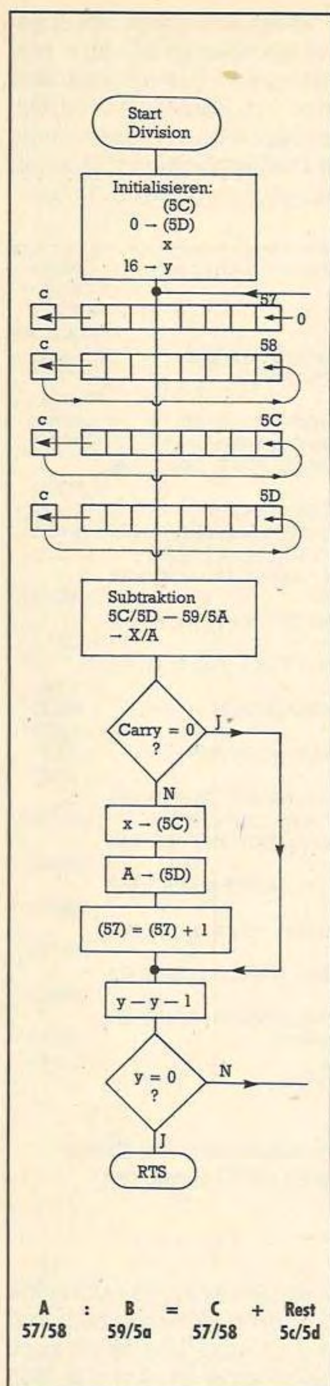


Bild 32. Flußdiagramm des Programms zur 16-Bit-Division

```
.
,5000  A2 00     LDX #00
,5002  86 5C     STX  5C
,5004  86 5D     STX  5D
,5006  A0 10     LDY #10
,5008  06 57     ASL  57
,500A  26 58     ROL  58
,500C  26 5C     ROL  5C
,500E  26 5D     ROL  5D
,5010  38        SEC
,5011  A5 5C     LDA  5C
,5013  E5 59     SBC  59
,5015  AA        TAX
,5016  A5 5D     LDA  5D
,5018  E5 5A     SBC  5A
,501A  90 06     BCC 5022
,501C  86 5C     STX  5C
,501E  85 5D     STA  5D
,5020  E6 57     INC  57
,5022  88        DEY
,5023  D0 E3     BNE 5008
,5025  60        RTS
----------------------------
.?
```

Programm 1. Die 16-Bit-Division

Faktor auf der rechten Seite wurde vom MSB an Bit für Bit durchgesehen. Jedesmal, wenn wir auf eine 1 gestoßen sind (hier waren nur Einsen), haben wir den links stehenden Faktor notiert. Dabei sind wir von mal zu mal um eine Stelle nach rechts gerückt, was mit dem Stellenwert des im rechten Faktor gerade betrachteten Bits zu tun hat. Das geschah so lange, bis alle Bits des rechten Faktors durchgearbeitet waren. Die sich auf diese Weise ergebende Kolonne wird dann addiert und führt zum Ergebnis. Vergleichen Sie, 240 ist wirklich binär 1111 0000.

Genauso wie hier beschrieben, arbeitet das Multiplikationsprogramm. Ein Unterschied tritt auf, nämlich daß nicht bis zum Schluß mit der Addition gewartet, sondern jede neue Zwischenzahl sofort addiert wird. Bild 29 zeigt die Beschreibung der Interpreterroutine:

Diese Routine hier abzudrucken, wäre reine Platzverschwendung. Schalten Sie einfach den SMON ein und verlangen Sie von ihm ein Disassemblerlisting ab $B357. Dort haben Sie dann für die weitere Besprechung alles parat. In Bild 30 finden Sie noch ein Flußdiagramm der UMULT-Routine.

Das Ergebnis der Multiplikation befindet sich in LSB/MSB-Form in den X/Y-Registern. Programm und Flußdiagramm wollen wir an einem Beispiel nachspielen. Dazu sollen die beiden Zahlen 321 und 65 (binär 0000 0001 0100 0001 und 0100 0001) miteinander multipliziert werden, was bekanntlich 20865 (binär 0101 0001 1000 0001) ergibt. Was Ihnen im Bild 31 als undurchdringlicher Bit-Dschungel entgegenstrahlt, ist das schrittweise Verfolgen des Programms im Computerformat, also binär.

In Bild 31 sind die Speicheradressen alle dezimal angegeben. Dort finden Sie zunächst die Ausgangslage. In Speicherstelle 40/41 steht die ganze Operation über unverändert die Zahl 321. In 113/114 finden Sie (wegen des LSB/MSB-Formates umgedreht als 114/113) unseren Faktor 65. Akku und Speicherstelle 93 stehen auf 16, dem Bitzähler. In das X- und Y-Register wurde eine Null eingelesen. Im Flußdiagramm ist diese Situation mit einer 1 gekennzeichnet. Ganz unten im Diagramm sehen Sie, daß der Bitzähler 93 erniedrigt und danach geprüft wird, ob er schon gleich Null sei. Daraus folgt, daß die große Schleife 16mal durchlaufen wird. Den ersten Durchlauf (gekennzeichnet durch kleine Buchstaben) verfolgen wir im einzelnen.

a) X-Register wird zur Bearbeitung in den Akku geschoben.
b) Mittels ASL wird das Bit 7 in die Carry-Flagge geschoben, was einen Carry-Inhalt von 0 bewirkt.
c) Der solchermaßen bearbeitete Akku-Inhalt (der sich hier nicht weiter verändert hat) geht wieder zurück ins X-Register.
d) Nun ist das Y-Register zur Bearbeitung dran. Es gelangt in den Akku.
e) Mittels ROL wandert nun das MSB des X-Registers aus dem Carry-Bit in die 0-Bit-Position des Akku
f) und alles zusammen wieder ins Y-Register. Insgesamt wird dadurch die 16-Bit-Zahl im X/Y-Register um eine Stellenzahl erhöht, was der Vorbereitung zur Addition dient. (Erinnern Sie sich bitte: Die Kolonne der Einzelergebnisse wird ja addiert). Im Diagramm (ohne Buchstabenkennzeichnung) schließt sich hier noch eine Prüfung auf einen eventuellen Überlauf an, der dann mit einer Fehlermeldung beantwortet wird.
g) Nun wird das MSB der Speicherstelle 113 nach links ins Carry geschoben. Das ist auch hier noch eine Null.
h) Anschließend wandert dieser Carry-Inhalt als Bit 0 in Speicherstelle 114. Bit 7 von 114 landet dafür im Carry. Auch hier wird auf diese Weise die ganze 16-Bit-Zahl 113/114 um ein Bit nach links geschoben und im nächsten Schritt – im

	58	57	5A	59	5D	5C	A	X	Y	C	
I a	01010001	10000011	00000001	01000001	00000000	00000000	–	00000000	00010000	–	Ausgangslage n. Init.
b		00000110								1	1. Linksschieben
c	10100011									0	2. Linksschieben
d, e					00000000	00000000				0	3./4. Linksschieben
f	10100011	00000110	00000001	01000001	00000000	00000000	11111110	10111111	00001111	1, 0	Ende der 1. Schleife
II	01000110	00001100	00000001	01000001	00000000	00000001	11111110	11000000	00001110	1, 0, 1, 0	Ende der 2. Schleife
III	10001100	00011000	00000001	01000001	00000000	00000010	11111110	11000001	00001101	1, 0	Ende der 3. Schleife
IV	00011000	00110000	00000001	01000001	00000000	00000101	11111110	11000100	00001100	1, 0, 1, 0	Ende der 4. Schleife
V	00110000	01100000	00000001	01000001	00000000	00001010	11111110	11001001	00001011	1, 0	Ende der 5. Schleife
VI	01100000	11000000	00000001	01000001	00000000	00010100	11111110	11010011	00001010	1, 0	Ende der 6. Schleife
VII	11000001	10000000	00000001	01000001	00000000	00101000	11111110	11100111	00001001	1, 0, 1, 0	Ende der 7. Schleife
VIII	10000011	00000000	00000001	01000001	00000000	01010001	11111111	00010000	00001000	1,1,0,1,0	Ende der 8. Schleife
IX	00000110	00000000	00000001	01000001	00000000	10100011	11111111	01100010	00000111	1, 0, 1, 0	Ende der 9. Schleife
X a	00001100	00000000			00000001	01000110	00000000	00000101		1, 0, 1, 1	
b	00001100	00000001	00000001	01000001	00000000	00000101	00000000	00000101	00000110		Ende der 10. Schleife
XI	00011000	00000010	00000001	01000001	00000000	00001010	11111110	11001001	00000101	1, 0	Ende der 11. Schleife
XII	00110000	00000100	00000001	01000001	00000000	00010100	11111110	11010011	00000100	1, 0	Ende der 12. Schleife
XIII	01100000	00001000	00000001	01000001	00000000	00101000	11111110	11100111	00000011	1, 0	Ende der 13. Schleife
XIV	11000000	00010000	00000001	01000001	00000000	01010000	11111111	00001111	00000010	1, 0	Ende der 14. Schleife
XV	10000000	00100000	00000001	01000001	00000000	10100001	11111111	01100000	00000001	1, 0, 1, 0	Ende der 15. Schleife
XVIa,	00000000	01000000			00000001	00000011	00000000	00000010		1,1,0,1,1	
b,	00000000	01000001	00000001	01000001	00000000	00000010	00000000	00000010	00000000		Ende der 16. Schleife = Endlage

Bild 33. 16-Bit-Division Schritt für Schritt am Beispiel 20867:321=65 Rest 2

Flußdiagramm wieder ohne Buchstaben – geprüft, ob da eine 1 oder eine 0 ins Carry-Bit geshiftet wurde. Wenn lediglich eine Null auftrat – wie hier –, dann springt das Programm sofort zum Herabzählen des Bitzählers 93. Tritt aber eine 1 auf, dann addiert sich der Inhalt von 40/41 zu X/Y.
i) Hier wird der Zustand der betroffenen Speicherstellen und Register nach dem ersten Schleifendurchlauf gezeigt.
Römisch II bis XVI zeigen nun jeweils den Zustand nach dem 2. bis 16. Durcharbeiten der großen Schleife. Wenn Sie verstehen möchten, was da passiert, sollten Sie versuchen, Bild 31 nur als Kontrolle zu verwenden und ansonsten mal selbst alle Schritte nachzuvollziehen.

44. 16-Bit-Division

Beim umgekehrten Weg, nämlich der Teilung von zwei 16-Bit-Zahlen, haben wir nicht so viel Glück: Ich konnte keine derartige Routine im Interpreter entdecken. Nun gibt es aber fast in jedem Lehrbuch der Maschinensprache die Vorstellung eines solchen Programms, so daß man sich das schönste aussuchen kann. Das Prinzip ist auch da dasselbe, wie wir es von der normalen Division gewohnt sind: Der Divisor wird Schritt für Schritt vom Dividenden abgezogen. In der Literatur [1] fand ich eine sehr kurze Routine, die ich Ihnen leicht modifiziert als Programm 1 vorstellen will.

In Bild 32 ist ein Flußdiagramm dieser Routine gezeigt und in Bild 33 lacht Ihnen wieder das Bit-Gewirr entgegen, das Sie schon von der Multiplikation her kennen, hier aber für die Division.

Damit Sie wissen, wo was hinein- oder herauskommt:

A	: B	= C	+ Rest
↑	↑	↑	↑
$57/58	$59/5A	$57/58	$5C/5D

An dem folgenden Beispiel soll der Programmverlauf getestet werden: Wir teilen 20867 durch 321. Dabei kommt nach Adam Riese heraus: 65, Rest 2.

In folgender Weise wird in die Speicherzellen die Aufgabe eingespeist:

20867	$57	1000	0011	LSB
	$58	0101	0001	MSB
321	$59	0100	0001	LSB
	$5A	0000	0001	MSB
Als Ergebnis findet man dann:				
65	$57	0100	0001	LSB
	$58	0000	0000	MSB
Rest 2	$5C	0000	0010	LSB
	$5D	0000	0000	MSB

Als Bit-Zähler dient hier das Y-Register.
b) Erstes Linksschieben des LSB mittels ASL. Dabei gelangt die 1 in das Carry-Bit.
c) Hineinrotieren der 1 aus dem Carry in das MSB mittels ROL.
d), e) Linksrotieren der 16-Bit-Zahl in $5C/5D, die jetzt noch 0 ist.
f) Situation am Ende der ersten Schleife. Der Bitzähler ist um 1 reduziert.

Im folgenden wird dann jeweils die Situation am Ende der Schleife gezeigt. Beim Berechnen der Differenz muß jeweils darauf geachtet werden, daß die Subtraktion einer Zahl als Addition des Zweierkomplements ausgeführt wird. Das haben wir in den Kapiteln 11 und 14 kennengelernt. Allerdings muß an dieser Stelle nochmal gesagt werden, daß die 1, die zum Einerkomplement hinzuaddiert wird, um das Zweierkomplement zu erhalten, das gesetzte Carry-Bit ist. Nun dürfte es für Sie eigentlich keine Probleme mehr geben, was das Nachvollziehen der Divisionsroutine betrifft.

Damit dürfen wir getrost die 16-Bit-Arithmetik abschließen. Alle vier Grundrechnungsarten können Sie jetzt programmieren. Weitere Rechenarten, wie Potenzieren, das Ziehen von Wurzeln, Logarithmen etc. bedingen ohnehin, daß die Argumente oder Ergebnisse keine Integerzahlen sind. Hier werden wir dann mit Fließkommaarithmetik arbeiten und den dazu vorgesehenen Interpreterroutinen.

```
1 REM *********************************    <250>
2 REM *                               *    <229>
3 REM *          PROGRAMM  2          *    <125>
4 REM *                               *    <231>
5 REM *   ERSTELLEN UND AUFRUF EINES  *    <186>
6 REM *        HILFSBILDSCHIRMES      *    <216>
7 REM *                               *    <234>
8 REM * HEIMO PONNATH    HAMBURG 1985 *    <082>
9 REM *********************************    <002>
10 PRINT CHR$(147):POKE 785,0:POKE 786,96:
   GOTO 30                                 <095>
15 REM -------- UP CURSOR SETZEN ------    <112>
20 POKE 211,SP:POKE 214,Z:SYS 58640:RETURN <163>
25 REM- ERSTELLEN DES HILFSBILDSCHIRMES-   <123>
30 Z=1:SP=1:GOSUB 20:PRINT"***************
   ************************"               <151>
40 Z=21:SP=1:GOSUB 20:PRINT"**************
   *************************"              <211>
50 Z=10:SP=7:GOSUB 20:PRINT"TEST FUER DIE
   VERSCHIEBUNG"                           <110>
55 REM ---- AUFRUF ZUM VERSCHIEBEN ----    <033>
60 A=USR(DUMMY)                            <195>
65 REM ---BILDSCHIRM NEU BESCHREIBEN---    <193>
70 GET A$:IF A$=""THEN 70                  <122>
80 PRINT CHR$(147):Z=2:SP=2:GOSUB 20:PRINT
   "JETZT SOLLTE DER ALTE BILDSCHIRM"      <092>
90 Z=4:SP=2:GOSUB 20:PRINT"UNTER DAS KERNA
   L-ROM GESCHOBEN SEIN"                   <150>
100 PRINT:PRINT:PRINT" -- JEDER{2SPACE}USR
    -AUFRUF HOLT DEN --"                   <003>
110 PRINT" -- HILFSBILDSCHIRM WIEDER .{3SP
    ACE}--"                                <068>
120 PRINT" -- AUCH IM DIREKT-MODUS{7SPACE}
    --"                                    <056>
130 PRINT:PRINT:PRINT"{2SPACE}PROBIEREN SI
    E MAL: A=USR(1) [RETURN]"              <050>
140 Z=19:SP=0:GOSUB 20:END                 <164>
```

© 64'er

Listing 2. Das Demo-Programm zur neuen Verschieberoutine. Vorher müssen Listing 3 und Programm 4 geladen werden.

Im Kapitel 32 haben wir ein Projekt gestartet, das dort eine Kopfzeile rückholbar unter den oberen ROM-Bereich verschob. Unser Wissen ist seither gestiegen und damit auch unsere Ansprüche. Eine Kopfzeile reicht nicht mehr, jetzt soll es ein ganzer Hilfsbildschirm sein, den wir erst in aller

45. Das Programmprojekt wird fortgeführt

Ruhe erstellen wollen, um ihn dann jederzeit abrufbar unter das Betriebssystem zu packen. Den Aufruf wollen wir wieder mit der USR-Funktion steuern. Diesmal soll aber so programmiert werden, daß der Hilfsbildschirm erhalten bleibt, man ihn also mehrfach einblenden kann. Über die Nützlichkeit einer solchen Routine braucht man sicherlich nicht viele Worte zu verlieren: Denken Sie da nur mal an Programme, die irgendwelche Tasten mit besonderen Funktionen belegen, für die Sie eine Gedächtnisstütze brauchen, oder...

Als Listing 2 ist ein kleines Demo-Programm abgedruckt, welches zuerst einen Bildschirm erstellt, dann die Routine »Verschieben« aufruft, den Bildschirm löscht und neu be-

```
.
,6000  A9 00     LDA #00        ,6021  A9 E8     LDA #E8
,6002  85 5F     STA   5F       ,6023  85 5A     STA   5A
,6004  A9 04     LDA #04        ,6025  A9 DB     LDA #DB
,6006  85 60     STA   60       ,6027  85 5B     STA   5B
,6008  A9 E8     LDA #E8        ,6029  A9 D1     LDA #D1
,600A  85 5A     STA   5A       ,602B  85 58     STA   58
,600C  85 58     STA   58       ,602D  A9 E7     LDA #E7
,600E  A9 07     LDA #07        ,602F  85 59     STA   59
,6010  85 5B     STA   5B       ,6031  20 BF A3  JSR A3BF
,6012  A9 E3     LDA #E3        ,6034  A9 40     LDA #40
,6014  85 59     STA   59       ,6036  8D 11 03  STA 0311
,6016  20 BF A3  JSR A3BF       ,6039  A9 60     LDA #60
,6019  A9 00     LDA #00        ,603B  8D 12 03  STA 0312
,601B  85 5F     STA   5F       ,603E  60        RTS
,601D  A9 D8     LDA #D8        ----------------------------
,601F  85 60     STA   60       .?
```

Listing 3. Erster Teil der Verschieberoutine

schreibt und schließlich mit einem weiteren USR den alten Bildschirm einblendet (vorher Listing 3 und 4 laden).

Von nun an können Sie immer - auch im Direktmodus - durch ein USR-Kommando diesen Bildschirm abbilden. Zum Programm in Kapitel 32 sind noch zwei Dinge zu bemerken, die hier geändert werden sollen. Erstens eine Frage: Ist Ihnen der Computer mal abgestürzt beim Aufruf des Programms? Die Wahrscheinlichkeit dafür ist ungefähr 1 : 60, wenn nämlich ein Interrupt stattfindet, während die Speicherstelle 1 geändert wird. Obwohl wir erst in den nächsten Folgen auf Interrupts eingehen werden, wollen wir die Wahrscheinlichkeit für so einen Absturz auf Null reduzieren. Eine andere Sache ist der Ort, an dem sich das Programm befand. Es hat sich nämlich herausgestellt, daß anscheinend die Nutzung dieses dort gewählten Speicherbereichs nicht ganz so problemlos ist. Bei einigen Anrufen wurde mir erzählt, daß zumindest der Anfang ab $02A7 bei bestimmten Konstellationen überschrieben wird. Deswegen legen wir unser Programm ganz unkonventionell nach $6000, von wo Sie es - das beherrschen Sie ja mit dem SMON inzwischen sicher - dorthin schieben können, wo es Ihnen gefällt. Allerdings müssen dann auch die USR-Adressen geändert werden. Aber auch das dürfte für Sie inzwischen kein Problem mehr sein.

Um diese immerhin schon 2000 Byte (1000 für den Bildschirm und noch mal 1000 für das Farb-RAM) zu verschieben, bedienen wir uns einer Interpreter-Routine, die auch beim Checksummer verwendet wird - der Blockverschiebe-Routine (Bild 34).

```
.
,603F  EA        NOP            ,6084  85 22     STA   22
,6040  A9 00     LDA #00        ,6086  A8        TAY
,6042  85 5F     STA   5F       ,6087  A5 5B     LDA   5B
,6044  A9 E0     LDA #E0        ,6089  E5 60     SBC   60
,6046  85 60     STA   60       ,608B  AA        TAX
,6048  A9 E8     LDA #E8        ,608C  E8        INX
,604A  85 5A     STA   5A       ,608D  98        TYA
,604C  85 58     STA   58       ,608E  F0 23     BEQ 60B3
,604E  A9 E3     LDA #E3        ,6090  A5 5A     LDA   5A
,6050  85 5B     STA   5B       ,6092  38        SEC
,6052  A9 07     LDA #07        ,6093  E5 22     SBC   22
,6054  85 59     STA   59       ,6095  85 5A     STA   5A
,6056  20 77 60  JSR 6077       ,6097  B0 03     BCS 609C
,6059  A9 E9     LDA #E9        ,6099  C6 5B     DEC   5B
,605B  85 5F     STA   5F       ,609B  38        SEC
,605D  A9 E3     LDA #E3        ,609C  A5 58     LDA   58
,605F  85 60     STA   60       ,609E  E5 22     SBC   22
,6061  A9 D1     LDA #D1        ,60A0  85 58     STA   58
,6063  85 5A     STA   5A       ,60A2  B0 08     BCS 60AC
,6065  A9 E7     LDA #E7        ,60A4  C6 59     DEC   59
,6067  85 5B     STA   5B       ,60A6  90 04     BCC 60AC
,6069  A9 E8     LDA #E8        ,60A8  B1 5A     LDA  (5A),Y
,606B  85 58     STA   58       ,60AA  91 58     STA  (58),Y
,606D  A9 DB     LDA #DB        ,60AC  88        DEY
,606F  85 59     STA   59       ,60AD  D0 F9     BNE 60A8
,6071  20 77 60  JSR 6077       ,60AF  B1 5A     LDA  (5A),Y
,6074  60        RTS            ,60B1  91 58     STA  (58),Y
------------------------------  ,60B3  C6 5B     DEC   5B
,6075  EA        NOP            ,60B5  C6 59     DEC   59
,6076  EA        NOP            ,60B7  CA        DEX
,6077  78        SEI            ,60B8  D0 F2     BNE 60AC
,6078  A5 01     LDA   01       ,60BA  68        PLA
,607A  48        PHA            ,60BB  85 01     STA   01
,607B  A9 35     LDA #35        ,60BD  58        CLI
,607D  85 01     STA   01       ,60BE  60        RTS
,607F  38        SEC            ----------------------------
,6080  A5 5A     LDA   5A       .?
,6082  E5 5F     SBC   5F
```

Listing 4. Zweiter Teil der Verschieberoutine

Name	**BLTUC**
Zweck	Verschieben von Speicherinhalten im Speicher
Adresse	$A3BF dez. 41919
Vorbereitungen	Quelle Startadresse nach $5F/60
	Endadresse+1 nach $5A/5B
	Ziel Endadresse+1 nach $58/59
Speicherstellen	$58-5B, $5F, $60, $22
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 34. BLTUC in Strichpunkten

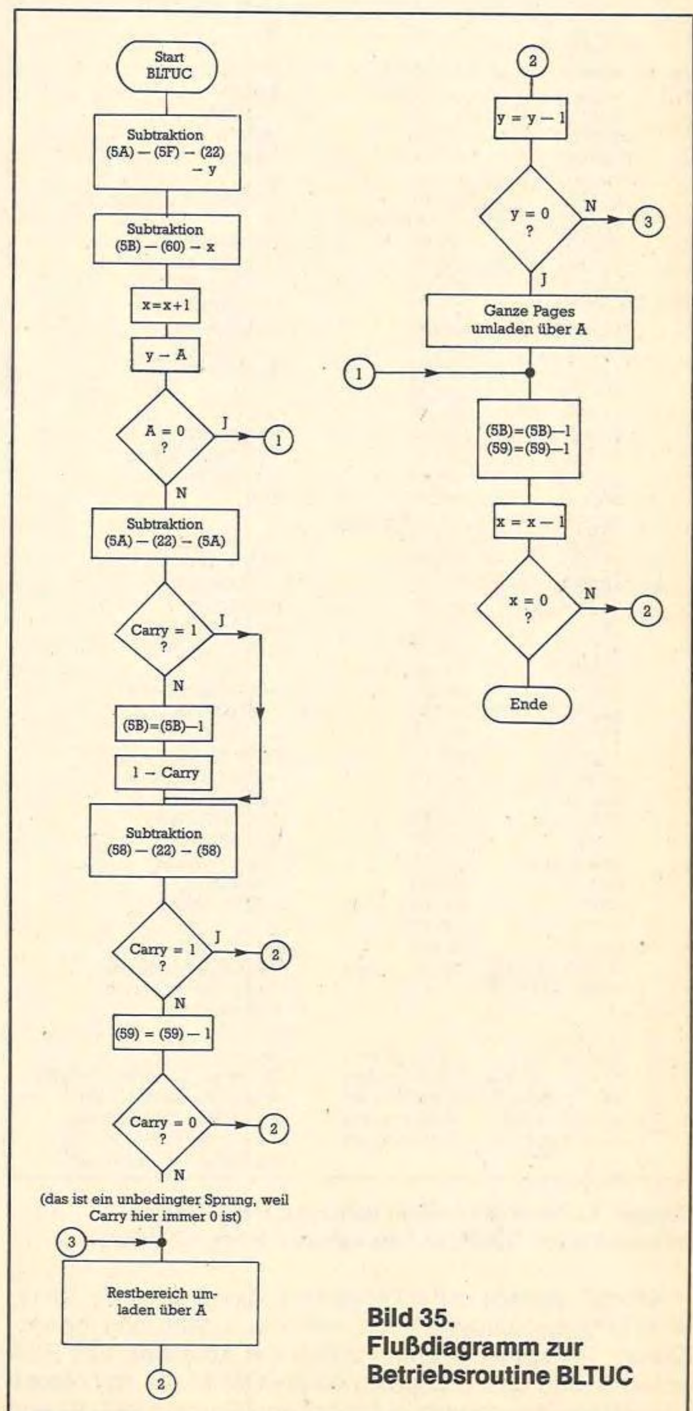


Bild 35. Flußdiagramm zur Betriebsroutine BLTUC

Startadresse	Format	Zahl
$AEA8	MFLPT	Pi
$B1A5	MFLPT	—32768
$B9BC	MFLPT	1
$B9C1	1-Byte-Integer	3
$B9C2	MFLPT	0.434255942
$B9C7	MFLPT	0.576584541
$B9CC	MFLPT	0.961800759
$B9D1	MFLPT	2.88539007
$B9D6	MFLPT	0.707106781 = SQR(1/2)
$B9DB	MFLPT	1.41421356 = SQR(2)
$B9E0	MFLPT	—0.5
$B9E5	MFLPT	0.693147181 = 1n2
$BAF9	MFLPT	10
$BDB3	MFLPT	99999999.9
$BDB8	MFLPT	999999999
$BDBD	MFLPT	1000000000
$BF11	MFLPT	0.5
$BF16	4-Byte-Integer	—100000000
$BF1A	— " —	10000000
$BF1E	— " —	—1000000
$BF22	— " —	100000
$BF26	— " —	—10000
$BF2A	— " —	1000
$BF2E	— " —	—100
$BF32	— " —	10
$BF36	— " —	—1
$BF3A	— " —	—2160000
$BF3E	— " —	216000
$BF42	— " —	—36000
$BF46	— " —	3600
$BF4A	— " —	—600
$BF4E	— " —	60
$BFBF	MFLPT	1.44269504 = 1/1n2
$BFC4	1-Byte-Integer	7
$BFC5	MFLPT	2.14987637E-05
$BFCA	MFLPT	1.43523140E-04
$BFCF	MFLPT	1.34226348E-03
$BFD4	MFLPT	9.61401701E-03
$BFD9	MFLPT	0.0555051269
$BFDE	MFLPT	0.240226385
$BFE3	MFLPT	0.693147186 = 1n2
$BFE8	MFLPT	1
$E08D	MFLPT	11879546
$E092	MFLPT	3.92767774E-08
$E2E0	MFLPT	1.57079633 = Pi/2
$E2E5	MFLPT	6.28318531 = 2*Pi
$E2EA	MFLPT	0.25
$E2EF	1-Byte-Integer	5
$E2F0	MFLPT	—14.3813907
$E2F5	MFLPT	42.0077971
$E2FA	MFLPT	—76.7041703
$E2FF	MFLPT	81.6052237
$E304	MFLPT	—41.3417021
$E309	MFLPT	6.28318531 = 2*Pi
$E33E	1-Byte-Integer	11
$E33F	MFLPT	—6.8473912E—04
$E344	MFLPT	4.85094216E—03
$E349	MFLPT	—0.0161117018
$E34E	MFLPT	0.034209638
$E353	MFLPT	—0.0542791328
$E358	MFLPT	0.0724571965
$E35D	MFLPT	—0.0898023954
$E362	MFLPT	0.110932413
$E367	MFLPT	—0.142839808
$E36C	MFLPT	0.19999912
$E371	MFLPT	—0.333333316
$E376	MFLPT	1
$E3BA	MFLPT	0.811635157
$E8DA - $E8E9	1-Byte-Integers	Tabelle der Farbcodes
$EB81 - $EBC1	— " —	Tastaturdecodierung: Einzelne Tasten
$EBC2 - $EC02	1-Byte-Integers	Tasten mit Shift
$EC03 - $EC43	1-Byte-Integers	Tasten mit Commodore-Taste
$EC78 - $ECB8	1-Byte-Integers	Tasten mit Control-Taste
$ECB9 - $ECE5	1-Byte-Integers	VIC-II-Chip-Registerwerte
$ECF0 - $ED08	1-Byte-Integers	Tabelle der LSBs der Bildschirm Anfangsadressen

Tabelle 16. Im ROM stehen nicht nur Programme, sondern auch Tabellen; hier einige wichtige Zahlen.

Wieder besteht unser Programm aus zwei Teilen. Im ersten wird der aktuelle Bildschirm nach oben geschoben. Dieser Teil speist lediglich zuerst die Adressen des Bildschirms und des Betriebssystem-ROM in die Abholspeicherstellen der danach aufgerufenen Routine BLTUC und wiederholt diesen Vorgang für die Bildschirmfarbspeicheradressen. Danach verstellen wir noch den USR-Vektor und kehren mit RTS ins Basic-Programm zurück (siehe Listing 3).

Komplexer ist der zweite Teil. Um nämlich die Informationen unter dem ROM lesen zu können, muß dieses ausgeschaltet werden. Leider läßt sich das Betriebssystem-ROM nur zusammen mit dem Basic-Interpreter ausschalten. $A3BF ist aber eine Interpreter-Routine! Da bleibt uns nichts anderes übrig, als diese Routine in unser Programm einzubauen, was uns die Gelegenheit gibt, sie uns mal etwas anzusehen. Als Bild 35 ist sie im Flußdiagramm abgebildet.

Listing 4 zeigt den zweiten Teil unseres Hilfsbildschirm-Programms.

Von $6040 an, wohin wir am Ende des ersten Teils den USR-Vektor gerichtet haben, wird zunächst wieder Quell- und Zielbereich in den Abholspeicherstellen spezifiziert und jeweils danach zuerst für den Bildschirm, dann für das Farb-RAM, das übernommene Unterprogramm angesprungen. Ab $6077 liegt dann das modifizierte Unterprogramm. Die Befehle SEI und CLI gehören zu den wenigen, die Sie erst noch kennenlernen. Sie sind es, die die Absturzwahrscheinlichkeit auf Null bringen. Jedenfalls wird zuerst das ROM aus und dafür RAM eingeschaltet. Ab $607F bis $60B9 befindet sich die Interpreter-Routine BLTUC. Darin wird zunächst die Länge des zu verschiebenden Bereichs berechnet, dann festgestellt, ob nur ganze Pages (Seiten) oder auch ein Restbereich verschoben werden soll. Falls ein solcher Restbereich vorhanden ist, wird auch seine Länge berechnet und zuerst dieser verschoben. Daran schließt sich das Verschieben der ganzen Pages an. Das X- und das Y-Register dienen dabei als Zähler.

Ab $60BB schließt sich wieder unsere eigene Routine an, in der wir die ROMs wieder einschalten. Auf diese Weise lassen sich noch mehrere Hilfsbildschirme unter ROM-Bereiche packen. Vielleicht überlegen Sie sich mal dazu einen Weg?

46. Die ROM-Bereiche als Datenquelle

Die ROM-Bereiche enthalten nicht nur ausgeklügelte Maschinenprogramme, sondern auch eine Menge Daten. Sollten Sie mal in die Verlegenheit kommen, beispielsweise die Zahl Pi im MFLPT-Format verwenden zu müssen, dann erfordert das einen ganz schönen Aufwand an Rechen- und Programmarbeit, oder Sie möchten bestimmte Texte wie beispielsweise eine Fehlermeldung verfügbar halten ... und so weiter. Viele von diesen Daten sind schon in der Firmware enthalten und wir werden im folgenden festhalten, wo sie sich befinden und welches Format man vorfindet. Sehen wir uns zunächst Zahlen an (Tabelle 16): Es existieren noch weitere Zahlentabellen in den ROM-Bereichen, die aber selten von Interesse sind. Ebenso wie Zahlen, findet man auch Texte im ROM als ASCII-Werte abgelegt (Tabelle 17)

Sollten Sie mal in die Verlegenheit kommen, solche Texte ausgeben zu wollen, dann legen Sie sie nicht noch mal in einer eigenen Texttabelle ab, sondern schöpfen Sie aus dem Fundus, den wir im ROM-Bereich fix und fertig haben.

Nun noch die Tabelle 18 mit den neuen Assembler-Befehlen.

[1] »Computerspiele und Wissenswertes Commodore 64«, Haar bei München: Markt & Technik Verlag, 1984. Das ist die von P. Lücke besorgte Übersetzung des amerikanischen Buches »More on the sixtyfour« und ist jedem Assembler-Programmierer warm zu empfehlen.

$A004	CBMBASIC
$A09E - $A19D	Texte der Basic-Befehlsworte (im letzten Byte ist jeweils Bit 7 gesetzt)
$A19E - $A327	Texte der Basic-Fehler- und System-Meldungen. (Im letzten Byte ist jeweils Bit 7 gesetzt)
$A364 - $A38A	Weitere System-Meldungen: OK, ERROR, IN, READY, BREAK. (Das letzte Byte ist jeweils 0)
$ACFC - $AD1D	Fehlermeldungstexte für INPUT: ?EXTRA IGNORED, ?REDO FROM START. (Das letzte Byte ist jeweils 0)
$E460	BASIC BYTES FREE
$E473	**** COMMODORE 64 BASIC V2 **** 64K-RAM-System
$ECE6	LOAD (Return) RUN (Return)
$F0BD - $F12B	Texte für Ein- und Ausgabe-Operationen
$FD10	CBM80

Tabelle 17. Diese Texte sind im ROM als ASCII-Werte abgelegt.

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beein-flussung von Flaggen
LSR	»Akkumulator«	1	1A	26	2	N,Z,C
	absolut	3	4E	78	6	N,Z,C
	0-page-absolut	2	46	70	5	N,Z,C
	absolut-X-indiz.	3	5E	94	7	N,Z,C
	0-page-X-indiz.	2	56	86	6	N,Z,C
ROL	»Akkumulator«	1	2A	42	2	N,Z,C
	absolut	3	2E	46	6	N,Z,C
	0-page-absolut	2	26	38	5	N,Z,C
	absolut-X-indiz.	3	3E	62	7	N,Z,C
	0-page-X-indiz.	2	36	54	6	N,Z,C
ROR	»Akkumulator«	1	6A	106	2	N,Z,C
	absolut	3	6E	110	6	N,Z,C
	0-page-absolut	2	66	102	5	N,Z,C
	absolut-X-indiz.	3	7E	126	7	N,Z,C
	0-page-X-indiz.	2	76	118	6	N,Z,C

Tabelle 18. Die neu besprochenen Assembler-Befehle

Die Assembler-Befehle haben wir jetzt bis auf vier noch offenstehende alle behandelt. Diese vier, die alle mit dem Interrupt-Handling zusammenhängen, sollen nun unser

47. Was sind Interrupts?

Thema sein. Wenn wir sie beherrschen, haben wir den ersten Schritt zum Meister der Assembler-Alchimie getan. Diese vier kleinen 1-Byte-Befehle öffnen uns eine geheime Pforte zu einem Universum an Programmier-Möglichkeiten, von dem wir bisher kaum zu träumen vermochten. Genug der Schwärmerei, erst kommt noch eine Menge Arbeit, die uns wohl mehrere Kapitel in Atem halten wird.

Zuvor noch eine Bemerkung: es gibt kaum ein Thema im Rahmen der Programmierung in Assembler, welches so penetrant häufig Abstürze provoziert, wie das nunmehr angesteuerte! Falls Sie noch keine Reset-Taste an ihrem Computer haben, wird es nun höchste Zeit. Diese nützlichen Schalter werden inzwischen schon so preiswert angeboten (sehen Sie mal in den Kleinanzeigenteil!), daß Sie zur Grundausstattung eines Assembler-Alchimisten zählen.

Unser Computer ist - solange er eingeschaltet ist - ständig mit irgendwelchen Tätigkeiten beschäftigt. Im Direktmodus hängt er beispielsweise meistens in einer Warteschleife und harrt der Eingaben, im Programm-Modus arbeitet er sich mit Hilfe der Interpreterschleife durch einen Basic-Befehlstext hindurch und so weiter. Nun werden Sie ja sicher schon festgestellt haben, daß er im Direktmodus auch den Cursor blinken läßt, in beiden Modi die TI$-Uhr weiterzählt und weitere Dinge erledigt, die anscheinend so nebenher passieren. Schon in Kapitel 8 aber haben wir einen Unterschied zwischen Mensch und Computer festgehalten: Der Mensch kann mehrere Dinge gleichzeitig bearbeiten, der Mikroprozessor ist nur fähig zu einer Aktion pro Zeiteinheit. Weil aber diese Zeiteinheiten so unfaßbar kurz sind (etwa eine Millionstel Sekunde), haben wir Benutzer den Eindruck der Gleichzeitigkeit.

Wenn dem aber so ist, wie macht es der Computer, daß er beispielsweise ein Programm abarbeitet und trotzdem die TI$-Uhr weiterzählt? Durch Unterbrechungen (interrupt = unterbrechen) der gerade ausgeübten Tätigkeit. Ein Beispiel aus dem täglichen Leben soll uns das illustrieren: Sie lesen gerade diesen Artikel, als das Telefon klingelt und ein Freund von Ihnen wissen möchte, was eigentlich Unterbrechungen sind. Während Sie es ihm erklären, fängt in der Küche der Teekessel schrill zu pfeifen an. Sie sagen Ihrem Freund, er möge sich einen Moment gedulden, gehen in die Küche und nehmen den Kessel vom Feuer. Dann kehren Sie ans Telefon zurück und beenden nach einer Weile das Gespräch. Nach dem Auflegen des Telefonhörers setzen Sie die Lektüre des Artikels fort, fest entschlossen, sich nun nicht mehr unterbrechen zu lassen. Kurze Zeit später klingelt jemand an der Tür. Sie lassen sich dadurch nicht stören.

Dieses Gleichnis gibt ziemlich genau wieder, was sich im Computer - nur bei millionenfacher Geschwindigkeit - bei Unterbrechungen abspielt. In Bild 36 ist das Schema des Ablaufes grafisch dargestellt. In gewisser Weise ähnelt das Ganze dem Abarbeiten von Unterprogrammsequenzen.

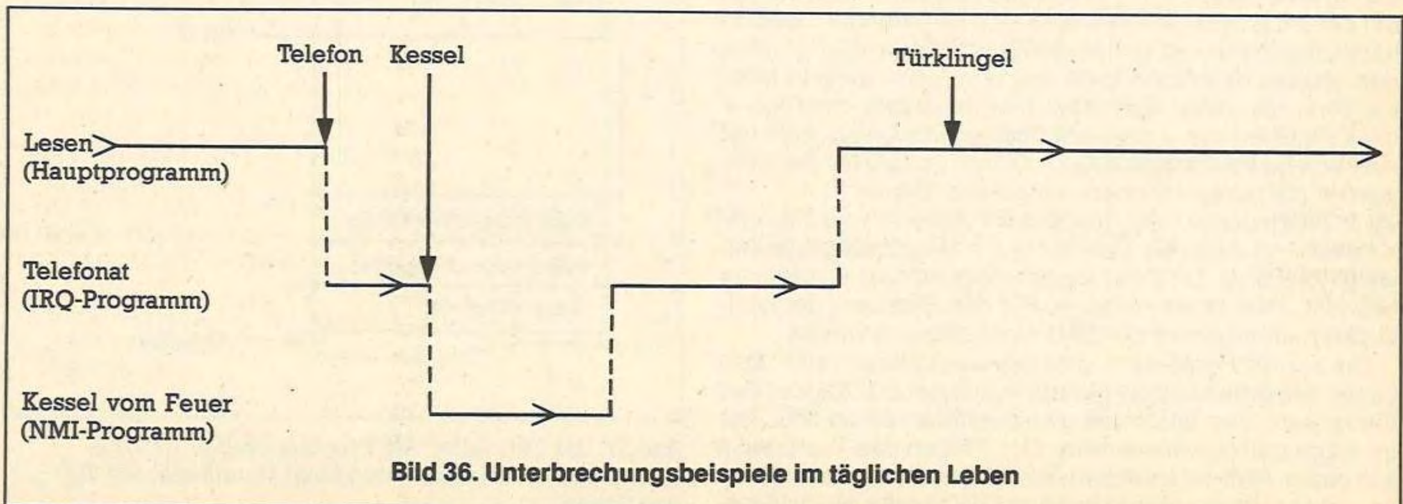


Bild 36. Unterbrechungsbeispiele im täglichen Leben

Weshalb programmiert man dann nicht einfach mittels einiger JSR-Aufrufe? Dafür hat L.A.Leventhal einen einleuchtenden Vergleich: »Ein Unterbrechungs-System entspricht etwa einer Telefonklingel. Sie läutet, wenn ein Anruf empfangen wird, so daß man den Hörer nicht laufend abnehmen muß, um festzustellen, ob sich jemand in der Leitung befindet.« (L.A.Leventhal, »6502 Programmieren in Assembler«, München te-wi Verlag, Seite 12-1). Unterbrechungen können dann angefordert und abgearbeitet werden, wenn sie nötig sind, im Gegensatz zu Unterprogrammen, die erst dann berücksichtigt werden, wenn der Programmzähler einen JSR-Befehl erfaßt. Um also schnell reagieren zu können, müßte man sehr oft in einem Programm eine Unterroutine anspringen, die auf gewisse Registerinhalte prüft und dann zur Bearbeitung verzweigt oder - bei Nichtvorliegen einer Bedingung - im normalen Programm weiterfährt. Das kostet unnötig Zeit und Speicherraum. Mancher Verkehr des Computers mit Peripherie erfordert so schnelle Reaktionen, daß diese nur geleistet werden können durch Unterbrechen des laufenden Programmes.

Ich denke, daß Sie nun die Notwendigkeit von Unterbrechungen erkennen. Fast jede CPU kennt solche Unterbrechungssysteme. Man kann sie charakterisieren durch die Beantwortung folgender Fragen:

1) Welche Unterbrechungs-Eingänge weist die CPU auf?
2) Wie reagiert die CPU auf eine Unterbrechung?
3) Wie bestimmt die CPU die Unterbrechungsquelle, wenn die Anzahl der Quellen größer ist als die Anzahl der Eingänge?
4) Kann die CPU zwischen wichtigen und weniger wichtigen Unterbrechungen unterscheiden?
5) Wie und wann wird das Unterbrechungssystem freigegeben oder gesperrt?

All diese Fragen werden wir für unseren Computer ergründen.

48. Das Unterbrechungssystem der CPU 6510/6502

Einige dieser Charakteristika sind schnell zu zeigen:

Zu 1: Unsere CPU hat genau 2 Eingänge für Unterbrechungen (wenn man RESET außer acht läßt, was wir im folgenden meist tun werden).

Zu 3: Natürlich gibt es weitaus mehr denkbare Unterbrechungsquellen als diese 2 Eingänge, weshalb per Software eine Registerabfrage (das sogenannte Polling) durchgeführt wird, um die Quelle festzustellen.

Zu 4: Zwischen wichtiger und nicht so wichtiger Unterbrechung kann unsere CPU unterscheiden durch die Priorität der beiden Eingänge. Wir haben eine sogenannte maskierbare Unterbrechung, genannt IRQ, welche per Befehl ignoriert (maskiert) werden kann und eine andere, nicht maskierbare, die daher auch NMI (not maskable interrupt = nicht maskierbare Unterbrechung) genannt wird. NMI hat eine höhere Priorität als IRQ und kann deshalb für die wichtigeren Aufgabenstellungen eingesetzt werden.

Zu 5: Freigegeben oder gesperrt werden kann die IRQ-Unterbrechung durch ein Sperrbit (auch Maskenbit genannt), welches sich als Bit 2 im Flaggen-Register des Prozessors befindet. Das ist die I-Flagge. Für den Empfang der NMI-Unterbrechung kann die CPU nicht gesperrt werden.

Um mal die Parallele zu unserem Beispiel zu zeigen: Das Lesen des Artikels ist die gerade stattfindende Tätigkeit des Computers. Die Telefonklingel signalisiert einen IRQ, der im folgenden bearbeitet wird. Das Pfeifen des Teekessels soll einem NMI entsprechen. Wenn dieser dann bearbeitet ist, geht es mit der Abarbeitung des IRQ weiter. Nach Beendigung des Telefonates wird das Unterbrechungs-Sperrbit gesetzt (sie nehmen sich vor, sich nicht mehr stören zu lassen) und mit der normalen Tätigkeit fortgefahren. Weil der nun folgende IRQ damit maskiert ist, wird das Türklingeln ignoriert.

Die Frage 2, nämlich wie unsere CPU auf eine Unterbrechung reagiert, blieb noch unbeantwortet. Nun soll sie behandelt werden:

a) Am Ende jedes Befehls überprüft die CPU automatisch den Zustand des Unterbrechungs-Systems. Wenn an einer der beiden Unterbrechungsleitungen eine Anforderung vorliegt und diese auch freigegeben ist, beginnt die Unterbrechung zu wirken.

b) Zunächst wird der Programmzählerinhalt in der Reihenfolge MSB, LSB auf den Stapel geschrieben. Danach wandert noch der Prozessorstatus auf den Stapel (siehe Bild 37).

c) Durch Setzen des Unterbrechungs-Sperrbits I werden weitere maskierbare Unterbrechungen (IRQ) unterbunden.

d) Nun holt sich die CPU aus einem Vektor ganz am Ende des Speichers eine Adresse, lädt diese in den Programmzähler und startet auf diese Weise ein Serviceprogramm, das dem auslösenden Anlaß Rechnung trägt. In der Tabelle 19 sind die zu den Unterbrechungsformen und zum RESET gehörigen Vektoren aufgeführt.

Bevor wir uns weiter mit den so angesteuerten Routinen befassen, wollen wir die 4 Befehle kennenlernen, die uns noch fehlen.

49. Schlüssel zur Unterbrechungsprogrammierung: CLI, SEI, RTI, BRK

Das Sperren der maskierbaren Unterbrechung IRQ und das Löschen der Maske erfolgt durch Setzen oder Löschen des Sperrbits im Prozessorstatus-Register. Dieses Bit, die I-Flagge, kann durch den Befehl CLI gelöscht werden. **CLI** kommt von »**CL**ear **I**nterrupt mask«, was bedeutet »lösche die Unterbrechungs-Maske«. Immer dann, wenn IRQs zugelassen sein sollen zur Bearbeitung durch den Mikroprozessor, muß damit die I-Flagge gelöscht werden. Wie Sie sehen, ist CLI ein 1-Byte-Befehl mit impliziter Adressierung. Er braucht genau 2 Taktzyklen zur Erledigung seiner Aufgabe.

Wenn wir später eigene Unterbrechungsroutinen schreiben, stehen wir oft vor der Frage, ob wir innerhalb unseres Unterbrechungsprogramms weitere Unterbrechungen zu-

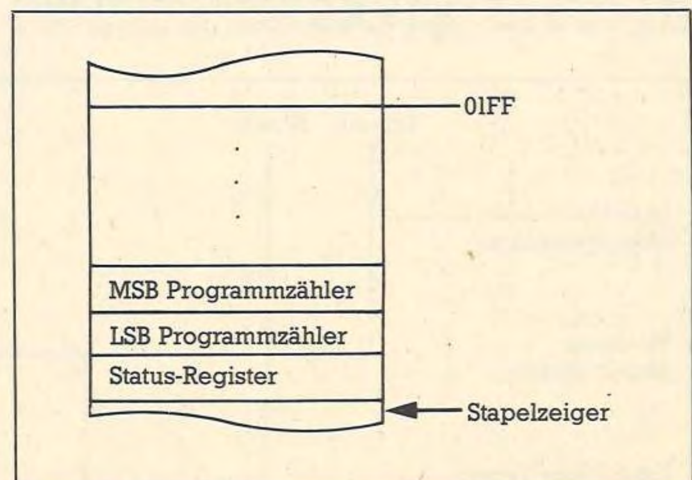


Bild 37. Die CPU rettet den Programmzähler und das Statusregister beim Eintreten einer Unterbrechung auf den Stapel

lassen wollen. Manchmal ist das wichtig, beispielsweise bei der Tastaturabfrage. Wie wir vorhin erwähnt haben, sperrt die CPU automatisch bei der Annahme von Unterbrechungen weitere IRQs durch Setzen der I-Flagge. Einer der ersten Befehle der eigenen Unterbrechungsroutine wird dann die Freigabe von Unterbrechungen sein durch Löschen der I-Flagge.

SEI bewirkt das Gegenteil von CLI. Der Befehl setzt die I-Flagge auf 1 (»**SE**t **I**nterrupt mask«) und verhindert, daß der Mikroprozessor weiteren IRQs seine Aufmerksamkeit schenkt. Das ist in den Fällen wichtig, in denen beispielsweise störungsfrei der Inhalt des Charakter-ROM gelesen werden soll oder während der Änderung von Speicherstellen, die die IRQ-Routine benutzt. Wie wichtig das Sperren von IRQs sein kann, haben Sie eventuell bemerkt, wenn Ihnen das Hilfsbildschirmprogramm aus Kapitel 32mal abgestürzt war. Seit der letzten Folge - wo wir die IRQs gesperrt haben - ist Ihnen das sicherlich nicht mehr passiert. Ebenso wie CLI ist SEI ein 1-Byte-Befehl mit impliziter Adressierung, und auch er braucht 2 Taktzyklen zur Bearbeitung.

Noch eine Bemerkung zum Verhindern der IRQs. Wir werden später sehen, was alles während der 60mal pro Sekunde aufgerufenen Unterbrechung erledigt wird. Jede Routine, die SEI verwendet, verbraucht Rechenzeit. Wenn sie so lange dauert, daß eine oder mehrere dieser regelmäßigen IRQs unterbunden werden, kann das unter Umständen zu Störungen von Programmabläufen führen. In solchen Fällen ist es sinnvoll, in die eigene Routine den Teil der regulären IRQ-Routine einzubauen, der im Programmablauf durch sein Fehlen Störungen verursacht. Meistens kann man aber durch gute Planung eines Programmes dieses Problem umgehen.

Unterbrechungsart	Vektor	Zieladresse
Maskierbare Unterbrechung (IRQ, BRK)	$FFFE/FFFF	65352 $FF48
Reset	$FFFC/FFFD	64738 $FCE2
Nichtmaskierbare Unterbrechung (NMI)	$FFFA/FFFB	65091 $FE43

Tabelle 19. Unterbrechungsvektoren und ihre Inhalte

RTI heißt »**R**e**T**urn from **I**nterrupt«, zu deutsch also: »kehre aus dem Unterbrechungsprogramm zurück.« Es entspricht in seinem Einsatz etwa dem RTS bei Unterprogrammrücksprüngen. Während RTS aber lediglich den alten Programmzählerinhalt vom Stapel holt (und noch eine 1 dazuaddiert), schafft RTI auch noch den alten Inhalt des Status-Registers vom Stapel zurück. Der genaue Ablauf ist wie folgt:

1) Alten Prozessorstatus vom Stapel wieder ins Status-Register schieben.
2) Stapelzeiger um 1 erhöhen
3) LSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
4) Stapelzeiger um 1 erhöhen
5) MSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
6) Stapelzeiger um 1 erhöhen.

Damit ist der Zustand vor der Unterbrechung wieder hergestellt. Auch die I-Flagge ist so automatisch wieder gelöscht, denn vor der Unterbrechung war sie sicher nicht gesetzt gewesen und der alte Status-Zustand ist ja jetzt wieder vorhanden.

RTI ist ebenfalls ein 1-Byte-Befehl mit impliziter Adressierung. Seine vollständige Bearbeitung dauert 6 Taktzyklen.

Bei eigenen Unterbrechungs-Routinen verwendet man häufig nicht RTI, sondern springt durch JMP an eine sinnvolle Stelle des normalen Unterbrechungsprogrammes. Auf diese Weise kann man dann die normalen Arbeitsgänge der vorprogrammierten Unterbrechung oder Teile davon noch ausführen lassen.

Den Befehl **BRK** (break=Software-Unterbrechung) haben wir schon verwendet. Er entspricht in seinem Einsatz etwa dem STOP-Befehl in Basic und dient wie jener Befehl dort hauptsächlich dem Testen von Programmen. Tatsächlich unterscheidet sich die Reaktion unserer CPU bei Auftreten eines BRK kaum von der bei einem IRQ. Folgendes passiert:

a) Der Programmzähler wird um 2 erhöht.
b) Bit 4 des Prozessorstatusregisters, die Break-Flagge B, wird auf 1 gesetzt.
c) Das MSB des Programmzählers wird auf den Stapel gebracht und der Stapelzähler um 1 heruntergezählt.
d) Dasselbe geschieht nun mit dem LSB des Programmzählers
e) und mit dem Statusregister.
f) Das Unterbrechungsmaskenbit, die I-Flagge, wird auf 1 gesetzt um IRQs zu sperren.
g) In den Programmzähler wird nun aus dem Vektor FFFE/FFFF dieselbe Adresse geladen, die auch bei IRQs benutzt wird. Damit startet nun das Programm, das diese Unterbrechung bearbeitet.

Sie sehen, daß der BRK-Befehl ein ziemlich komplizierter Geselle ist. Zwar handelt es sich wieder um einen 1-Byte-Befehl mit impliziter Adressierung, aber er benötigt immerhin sieben Taktzyklen, um all diese Arbeit zu bewältigen.

Wir haben BRK bisher immer zur Programmunterbrechung mit nachfolgender Registeranzeige durch den SMON eingesetzt. Der SMON ist - wie fast jeder Monitor - so programmiert, daß ein BRK zur Registeranzeige führt. Das ist natürlich sinnvoll beim Einsatz von BRK zur Fehlersuche. In dem Moment, wo ein BRK vom Prozessor bearbeitet wurde, kann nur durch die gesetzte B-Flagge von einem IRQ unterschieden werden. Es ist manchmal nötig, schon zu diesem Zeitpunkt diesen Unterschied festzustellen. Deshalb verwendet man den nachfolgend beschriebenen Test zu diesem Zweck:

PLA
in den Akku wird das zuletzt auf den Stapel geschobene Prozessorstatus-Register geholt.

PHA
und sogleich wieder zurückgeschoben

AND #$10
durch die AND-Verknüpfung mit der Binärzahl 0001 0000 kann eine eventuell vorhandene B-Flagge isoliert werden.

BNE BREAK
Falls eine B-Flagge gesetzt war, ist der Akku ungleich 0 und die Bearbeitung verzweigt zum von uns konstruierten BREAK-Programm. War der Akku nach dieser AND-Verknüpfung gleich 0, dann erfolgt keine Verzweigung und es handelt sich um einen IRQ, zu dessen Bearbeitung nun zu springen ist.

Es gibt noch eine andere - gebräuchlichere - Möglichkeit, zwischen einem BRK und einem IRQ zu unterscheiden, die allerdings erst zu einem späteren Zeitpunkt des computerinternen Unterprogrammes erfolgt. Von dieser zweiten Möglichkeit wird im SMON Gebrauch gemacht und wir werden sie nachher auch kennenlernen.

Natürlich kann der BRK-Befehl auch zu anderen Zwecken als zur Registeranzeige durch einen Monitor verwen-

wendet werden. Es kommt immer darauf an, welches Service-Programm wir dem Computer anbieten. Springt man aus so einem Service-Programm mittels RTI zurück ins Hauptprogramm, dann muß man berücksichtigen, daß der Programmzähler vor der Sicherung auf dem Stapel um 2 erhöht worden ist. Manchmal sind deshalb noch Korrekturen des Programms nötig.

Ich hoffe, daß Sie bisher dieses Kapitel nicht zu frustrierend fanden, denn ständig ist die Rede vom eigenen Unterbrechungs-Programm und dabei wissen Sie - außer durch BRK - noch gar keine Möglichkeit, einen IRQ oder NMI auszulösen, und Sie sind sicher noch sehr vorsichtig mit dem Gedanken an eigene Unterbrechungs-Routinen, weil Ihnen ja noch unbekannt ist, wie die normale Firmware Unterbrechungen behandelt. Keine Angst: All das werden wir noch klären. Betrachten Sie diesen Teil zum Thema Unterbrechungen vielleicht mehr wie ein Handbuch, in dem Sie dann, wenn Ihr Verständnis gestiegen ist, noch mal stöbern können.

Wir haben bisher nur betrachtet, wie unsere CPU reagiert, wenn an einem der beiden Unterbrechungs-Eingänge (IRQ und NMI) eine Unterbrechungs-Anforderung vorliegt. Um nun aber selbst ins Geschehen eingreifen zu können, ist es nötig zu wissen, wie diese Anforderung dorthin gelangt. Das erfordert von uns die Beschäftigung mit anderen Computerbausteinen als der CPU, die bisher im Mittelpunkt unseres Interesses stand.

50. Woher kommen die Unterbrechungs-Anforderungen?

Quellen für Unterbrechungen können viele genannt werden: Diskettenstation, Datasette, Drucker, Modem, Schaltelemente und so weiter. Um aber eine gewisse Übersicht zu bekommen, sollte man unterscheiden zwischen primären und sekundären Unterbrechungsquellen. Das soll kurz erläutert werden: Die Diskettenstation beispielsweise ist über den seriellen Port mit dem Computer verbunden. Dieser wiederum steht in direktem Kontakt zu 2 Bausteinen, den CIAs. Erst diese CIAs stehen in direktem Kontakt zur CPU. Alle Unterbrechungs-Quellen, die direkt Signale an die beiden Unterbrechungseingänge unserer CPU senden, sollen künftig »primäre« Quellen genannt werden, die anderen, die nur über solch eine primäre Quelle Unterbrechungs-Anforderungen stellen, werden von uns als »sekundäre« Quellen bezeichnet. Weil wir irgendwo einen Schnitt machen müssen - um nicht völlig auszuufern in der Erklärung von peripheren Geräten (das soll anderen, in diser Sache kompetenteren Autoren, überlassen bleiben) - werden wir uns im folgenden auf die primären Unterbrechungsquellen beschränken. Da bleibt aber noch mehr als genug zu tun übrig und deshalb soll auch nur eine Auswahl dieser Primärquellen detailliert behandelt werden.

Welches sind nun die primären Unterbrechungsquellen? Hier sind sie aufgeführt:

1) Der VIC-II-Chip (MOS 6566/6567 Video Interface Controller)
2) Die beiden CIAs (MOS 6526 Complex Interface Adapter)
3) Die RESTORE-Taste
4) Der Expansion-Port
5) RESET (paßt hier nicht ganz her, woanders aber auch nicht besser)

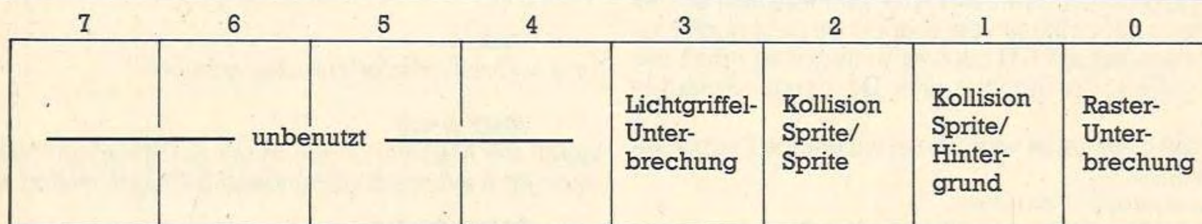
Den Expansion-Port werden wir nicht behandeln und einen RESET nur ziemlich kurz betrachten, weil es sich dabei eigentlich nicht um eine Unterbrechung im bisher definierten Sinn handelt.

51. Der VIC-II-Chip als Unterbrechungsquelle

Soweit ich feststellen konnte, kommt der VIC-II-Chip in bezug auf unsere CPU nur als Anforderer von maskierten Unterbrechungen (IRQ) in Frage. Die Handhabung seiner Unterbrechungs-Anforderungen geschieht im VIC-II-Chip durch zwei Register. Vier Ereignisse sind eingeplant, deren Eintreten zur Unterbrechung führen kann:

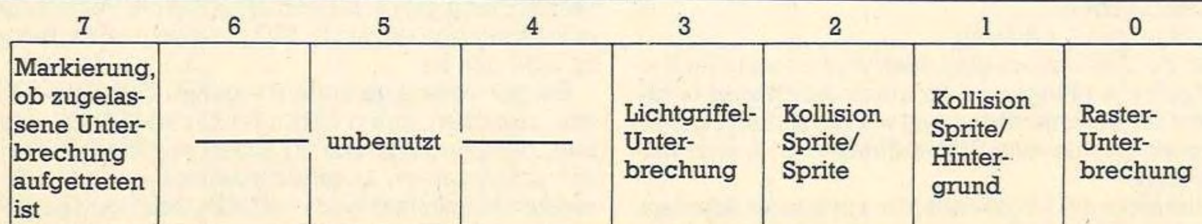
1) Rasterzeilen-Unterbrechung
2) Kollision eines Sprite mit Hintergrund
3) Kollision von Sprites untereinander
4) Lichtgriffel-Unterbrechung.

Die ersten drei Auslöser werden wir uns in kommenden Folgen genau ansehen und dabei vielerlei interessante Möglichkeiten feststellen. Die Option, die der Lichtgriffel bietet, wird nicht behandelt werden: Meine Kenntnisse auf diesem Sektor sind nur gering (nobody is perfect).



7	6	5	4	3	2	1	0
— unbenutzt —				Lichtgriffel-Unterbrechung	Kollision Sprite/Sprite	Kollision Sprite/Hintergrund	Raster-Unterbrechung

Bild 38. Das Interrupt-Enable-Register (53274 = $D01A) des VIC-II-Chip



7	6	5	4	3	2	1	0
Markierung, ob zugelassene Unterbrechung aufgetreten ist	— unbenutzt —			Lichtgriffel-Unterbrechung	Kollision Sprite/Sprite	Kollision Sprite/Hintergrund	Raster-Unterbrechung

Bild 39. Das Interrupt-Latch-Register (53273 = $D019) des VIC-II-Chip

Das sogenannte Interrupt Enable Register (Unterbrechungs-Zulassungs-Register) des VIC-II-Chips ist Register 26. Es befindet sich in der Speicherstelle 53274 ($D01A) (siehe Bild 38).

In diesem Register wird festgelegt, ob eines - oder mehrere - der vier möglichen auslösenden Ereignisse eine Unterbrechungsanforderung an den Mikroprozessor senden soll. Jedem Ereignis ist ein Bit zugeordnet. Ist dieses Bit gleich 1, dann ist die Unterbrechung freigegeben, ist es gleich 0, dann liegt eine Sperrung vor. Die Zuordnung der Bits ist wie folgt:

Bit 0 Rasterzeilen-IRQ
Bit 1 Sprite/Hintergrund-Kollision
Bit 2 Sprite/Sprite-Kollision
Bit 3 Lichtgriffel-IRQ
Bits 4 bis 7 sind ungenutzt und haben immer den Wert 1.

Das Register 25 wird Interrupt Latch Register genannt, was etwa zu übersetzen wäre mit »Unterbrechungs-Einrast-Register« (siehe Bild 39). Der englische Ausdruck »latch«, der nur umschreibend oder sehr technisch übersetzt werden kann, beschreibt eigentlich recht genau, was in diesem Register geschieht. Ein »latch« ist nämlich so etwas wie ein Schnappriegel, also ein Riegel, der bei der Betätigung einrastet. Wenn eines der 4 möglichen Ereignisse eintritt, schnappt im dazugehörigen Bit dieses Registers der Inhalt auf 1. Die Bit-Zuordnung ist die gleiche wie in Register 26. Aber das Bit 7 hat hier noch eine Bedeutung: Ist eines der Bits 0 bis 3 auf 1 gesetzt und das dazugehörige Ereignis in Register 26 auch zur Unterbrechung zugelassen (also auch dort gleich 1), dann taucht in Register 25, Bit 7 eine 1 auf. So kann durch einfaches Lesen dieses Bits festgestellt werden, ob ein IRQ durch den VIC-II-Chip ausgelöst wurde.

Will man in diesem Register ein gesetztes Bit löschen, muß man - und das ist außergewöhnlich - eine 1 in die Bitposition schreiben.

Mit Recht erwarten Sie nun eigentlich eine Anwendung des bisher gelernten. Bei Unterbrechungsprogrammen ist es aber dringend nötig, immer den gesamten Komplex im Auge zu haben. Ich habe mich daher entschlossen, zuerst alles zu erklären und dann Anwendungsmöglichkeiten vorzustellen. Ihre Geduld wird auf eine harte Probe gestellt, aber ich hoffe, daß Sie später feststellen, daß es sich gelohnt hat, etwas zu warten.

52. Die beiden CIA-Bausteine als Unterbrechungsquellen

An sich sind die beiden CIAs in unserem Computer völlig identisch. Sie werden aber unterschiedlich eingesetzt. Sehen wir uns zunächst einmal an, was beiden in bezug auf Unterbrechungen gemeinsam ist, um danach die Unterschiede festzuhalten. Die Unterbrechungs-Steuerung geschieht in Register 13 dieser Bausteine. Dieses Register hat 2 Funktionen: Es bestimmt, ob eine Unterbrechungsanforderung an die CPU gesandt werden soll, und es stellt fest, ob ein Ereignis stattgefunden hat, das zur Unterbrechung führen kann. Die Bedienung dieses Registers ist demzufolge auch etwas unübersichtlich, aber wir haben schon ganz andere Probleme gemeistert.

Sehen wir uns aber zuerst einmal an, welche Ereignisse vom Standpunkt eines CIA-Bausteines als Unterbrechungskriterium dienen können:

1) Unterlauf der Uhr A
2) Unterlauf der Uhr B
3) Die interne Uhr hat eine Alarmzeit erreicht
4) Am SP-Eingang (hängt mit dem seriellen Port zusammen) ist ein bestimmter Zustand erreicht
5) An einem Eingang namens FLAG ist ein bestimmter Zustand erreicht.

Die Ereignisse 4 und 5 werden wir ebenfalls im weiteren weitgehend ausklammern.

Nun zum Register 13, dem Unterbrechungs-Kontroll-Register (siehe Bild 40).

Auch hier gehört zu jedem Ereignis ein Bit. Dabei - um Wiederholungen zu vermeiden - ist die Zuordnung schon durch die eben angegebene Ereignisaufzählung gegeben. Ziehen Sie von der vorangestellten Nummer immer eine 1 ab und Sie haben die Bitnummer. Die Bits 5 und 6 sind unbenutzt. Bit 7 hat eine dreifache Funktion, die eng mit den anderen Bitinhalten verknüpft ist. Sehen wir uns das mal der Reihe nach an:

Lesen des Registers

Sind Unterbrechungsereignisse aufgetreten, dann sind die dazugehörigen Bits auf 1 gesetzt. Bit 7 ist gleich 1, wenn mindestens ein solches Ereignis stattgefunden hat und außerdem dieses Ereignis als Unterbrechungsauslöser freigegeben ist. Auf diese Weise kann - ähnlich wie beim VIC-II-Chip-Register 25 - festgestellt werden, ob die Unterbrechung durch einen der beiden CIAs angefordert wurde. Im Unterschied aber zum VIC-II-Register wird Register 13 durch das Lesen gelöscht. Braucht man den Inhalt also noch, sollte man ihn irgendwo zwischenspeichern.

Schreiben in das Register

Bit 7 = 0 erzeugt Sperren.

Das erkennt man am besten an einem Beispiel. Nehmen wir an, wir möchten die Unterbrechung sperren, die durch einen Unterlauf von Uhr A erzeugt werden kann. Das betrifft das Bit 0. Wir schreiben in das Register 13 folgende Zahl: 0000 0001

Wie Sie sehen, ist das Bit 7 gleich 0. Die 1 in Bit 1 bewirkt die Sperrung. Durch die Nullen in den anderen Bits wird bewirkt, daß die anderen Unterbrechungs-Ereignisse nicht beeinflußt werden. Wollten wir alle sperren, dann müßten wir einschreiben: 0001 1111

Auf diese Weise können selektiv einzelne Unterbrechungen durch Einschreiben der 1 bei gelöschtem Bit 7 gesperrt werden.

Bit 7 = 1 erzeugt Freigabe.

Auch hier wieder ein Beispiel. Wenn wir ganz gezielt Unterbrechungen durch Unterlauf der Uhr A freigeben wollen,

7	6	5	4	3	2	1	0
Mehr-funktions-Bit	— unbenutzt —		FLAG-Eingang	SP-Eingang	ALARM bei interner Uhr	Unterlauf Uhr B	Unterlauf Uhr A

Bild 40. Genereller Aufbau der Unterbrechungs-Kontroll-Register (13) der beiden CIA-Bausteine

müssen wir die folgende Zahl in Register 13 schreiben:
1000 0001
Bit 7 (gleich 1) zeigt an, daß diejenigen Unterbrechungen freizugeben sind, deren Bits auf 1 gesetzt sind. Alle anderen Unterbrechungen, wo also in der dazugehörigen Bitposition der einzuschreibenden Zahl eine 0 steht, bleiben unverändert.

Ein wichtiger Unterschied zwischen den beiden CIAs ist der, daß der Unterbrechungsausgang von CIA 1 mit dem IRQ-Eingang der CPU verbunden ist, wohingegen der entsprechende Ausgang von CIA2 an den NMI-Eingang unseres Mikroprozessors führt. Daher löst der CIA 1 nur IRQs aus, er wird manchmal deshalb auch IRQ-CIA genannt. Der andere ist dann der NMI-CIA, weil er nur NMIs anfordern kann.

53. Der IRQ-CIA

Das Register 13 des IRQ-CIA (der die Speicherstellen 56320 bis 56335 belegt), liegt in Zelle 56333 ($DC0D). Die einzelnen Bits sind wie folgt zugeordnet:

Bit 0 Unterlauf Uhr A
Von hier kommt der IRQ, der 60mal pro Sekunde stattfindet zur Tastaturabfrage, zum Weiterstellen der TI$-Uhr etc.
Bit 1 Unterlauf Uhr B
Spielt bei Kassettenoperationen und dem seriellen Port eine Rolle.
Bit 2 ALARM bei interner Uhr.
Spielt beim Zufallszahlengenerator (RND(0)) eine Rolle.
Bit 3 Hier kommen durch den User-Port Unterbrechungs-Anforderungen.
Bit 4 ist verbunden mit dem seriellen Port und der Kassetten-Lese-Leitung.

54. Der NMI-CIA

Ebenso kurz und schmerzlos wie beim CIA 1 soll auch das besondere am CIA 2, dem NMI-CIA (er belegt den Speicher von 56576 bis 56831) vorgestellt werden. Sein Register 13 findet sich in Speicherstelle 56589 ($DD0D). Die Bits 0 und 1 (Unterläufe der beiden Uhren) spielen beim Senden beziehungsweise Empfangen von Daten über die RS232C-Schnittstelle eine Rolle, Bit 2 (ALARM) wird nicht verwendet, Bit 3 ist direkt mit dem User-Port verbunden ebenso wie Bit 4. Der NMI-CIA wird uns in seiner normalen Funktion nicht mehr beschäftigen.

55. Die RESTORE-Taste und ein kleines Testprogramm

Die RESTORE-Taste ist direkt mit dem NMI-Eingang unseres Mikroprozessors verbunden. Das ermöglicht es uns, durch einfaches Drücken dieser Taste jederzeit ins Geschehen einzugreifen, ohne uns um Details kümmern zu müssen, ob sich der Computer gerade im Direkt- oder im Programm-Modus befindet und so weiter. Denn NMI hat die höchste Priorität der Unterbrechungen.

Ein kleines Testprogramm soll Ihnen hier noch vorgestellt werden, das Sie vielleicht aber noch nicht ganz verstehen werden, weil wir erst in der nächsten Folge die eingebauten Serviceprogramme kennenlernen werden. Schalten Sie also den SMON ein und geben Sie das Listing 5 ein (ab $6000):

Am besten speichern Sie nun das Programm und schalten dann mit dem SMON-Kommando M 0318 die Anzeige der Bytes ab $0318 ein. Dort steht in den beiden ersten Speicherzellen 47 und FE. Mit dem Cursor fahren Sie in diese Zeile und ändern den Inhalt in 00 und 60, also unsere Programmstartadresse in der LSB/MSB-Form. Nach einem RETURN läuft nun jede NMI-Anforderung über unser Programm. Nun können Sie es ausprobieren, indem Sie mal die RESTORE-Taste drücken. Es genügt völlig, alleine diese Taste zu betätigen. Das wirkt – sichtbar durch die Änderung der Rahmenfarbe – in jedem Modus und jederzeit. Eine kleine Merkwürdigkeit ist, daß manchmal etwas Geduld aufzubringen ist, bis man die Wirkung sieht. Ich vermute, daß der NMI so schnell erledigt wird, daß sich mehrere NMIs pro Tastendruck ereignen. Man müßte sich noch eine kleine Routine überlegen, die die Wirkung etwas verzögert, denn 2 solche EOR-Kommandos nacheinander heben sich gegenseitig auf. Tabelle 20 zeigt Ihnen die Unterbrechungsbefehle.

6000	PHA	mit diesen Befehlen retten wir Akku und Register auf den Stapel.
6001	TXA	
6002	PHA	
6003	TYA	
6004	PHA	
6005	LDA #$7F	0111 1111 ist das in binär.
6007	STA $DD0D	Dadurch werden alle NMIs, die vom CIA 2 kommen könnten, gesperrt. Erinnern Sie sich: Bit 7 ist Null beim Schreiben, also Sperrfunktion.
600A	LDY $DD0D	Lesen des Registers 13 löscht dieses und zeigt uns, ob die NMI-Anforderung von dort kam.
600D	BMI $601A	falls NMI-Anforderung vom CIA 2 kam, wird verzweigt
600F	LDA $D020	ansonsten kommt der NMI von der RESTORE-Taste, und in den Akku wird die Rahmenfarbe eingeladen
6012	EOR #$0E	Ausgehend davon, daß als Rahmenfarbe 14 vorliegt, wird diese exklusiv geORDERt zu Null. Ist die Rahmenfarbe 0, dann wird sie wieder 14.
6014	STA $D020	Einschreiben des neuen Farbwertes
6017	JMP $FEBC	Sprung in den Rest der normalen NMI-Routine
601A	JMP $FE72	Sprung in die normale NMI-Routine für den Fall, daß die Anforderung durch den NMI-CIA kam.

Listing 5. Ein kleines Testprogramm demonstriert die Wirkung einer Unterbrechung: Durch Drücken der RESTORE-Taste wird die Rahmenfarbe geändert.

56. Der normale Verlauf eines IRQ

Wir hatten bereits festgestellt, daß eine IRQ-Anforderung (nach dem Retten des Programmzählers und des Prozessorstatus-Registers, sowie dem Setzen der I-Flagge) den Inhalt des Vektors $FFFE/FFFF in den Programmzähler holt. Dort steht die Adresse $FF48(dez. 65352) und deshalb startet nun das dort im ROM verankerte Programm, welches wir uns nun im einzelnen ansehen werden (alle Adressen als Dezimalzahlen, in Bild 41 finden Sie das Flußdiagramm dazu).

```
65352  PHA    Zunächst werden der Akku und
       TXA    die Register X und Y auf den
       PHA    Stapel geschoben
       TYA
       PHA
```

Trickreich sind die beiden folgenden Befehle, mit denen das zu Beginn durch die CPU gerettete Statusregister gelesen wird:

```
       TSX        Stapelzeiger ins X- Register
       LDA 260,X  Einladen des Status-Registers
```

Nun wird geprüft, ob die BRK-Flagge gesetzt ist. Wenn das der Fall ist, dann ist der Auslöser ein BRK gewesen, ansonsten ein IRQ:

	AND #16	Isolieren der BRK-Flagge
	BEQ 65368	Wenn keine BRK-Flagge, dann überspringen des nächsten Befehls.
65365	**JMP (790)**	Falls BRK
65368	**JMP (788)**	Falls IRQ

Den vorletzten Sprungbefehl werden wir bei der BRK-Behandlung verfolgen. Interessant für uns ist jetzt der indirekte Sprung bei 65368. Der Vektor 788/789 ($314/315) liegt im RAM! Damit können wir ihn auf eigene Routinen verstellen. Genau hier ist der Ansatzpunkt für nahezu alle Eingriffe in die Unterbrechungsbehandlung. Der voreingestellte Wert in diesem Vektor ist die Adresse 59953 ($EA31). Das dort angesiedelte Programm wird im Normalfall 60mal in der Sekunde ausgeführt:

59953	**JSR 65514**	Das ist ein Kernel-Sprungbefehl zur Routine UDTIM bei 63131.

Befehls-wort	Adressie-rung	Byte-zahl	Code Hex	Code Dez	Takt-cyclen	Beeinflussung von Flaggen
CLI	implizit	1	58	88	2	I-Flagge
SEI	implizit	1	78	120	2	I-Flagge
RTI	implizit	1	40	64	6	alle Flaggen
BRK	implizit	1	00	0	7	B-Flagge vor dem Schieben auf den Stapel, I-Flagge danach

Tabelle 20. Die Daten zu den letzten Assembler-Befehlen

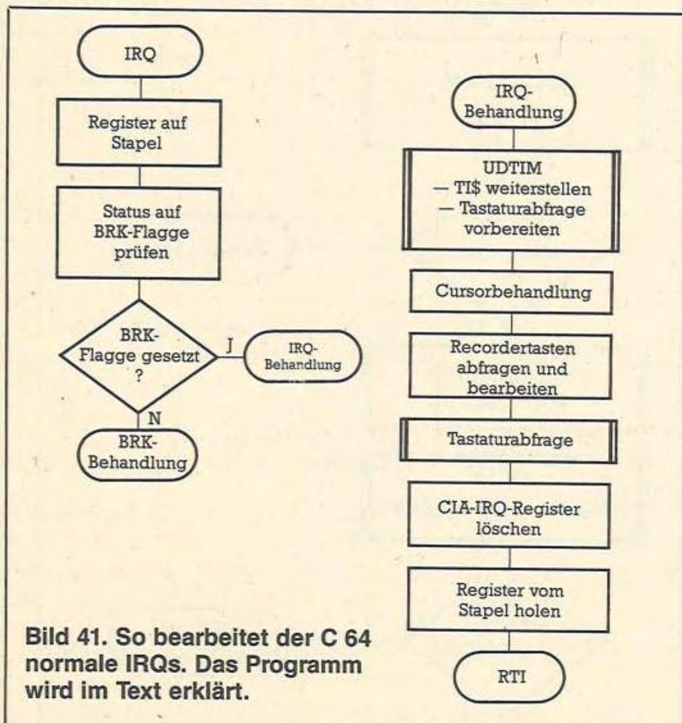


Bild 41. So bearbeitet der C 64 normale IRQs. Das Programm wird im Text erklärt.

In diesem Unterprogramm wird zuerst die Uhr TI$ weitergestellt und dann die Tastaturabfrage vorbereitet.

59956 bis 60000		In diesem Programmteil erfolgt die Cursor-Behandlung.
60001 bis 60026		Anschließend wird abgefragt, ob eine Recordertaste gedrückt ist und entsprechende Flaggen bearbeitet.
60027	**JSR 60039**	Dieses Unterprogramm dient zur Tastaturabfrage.

Auch in dieser Routine tritt übrigens ein indirekter Sprung nach einem RAM-Vektor auf (655/656 = $28F/290), der normalerweise auf 60232 zeigt, aber auch auf eine eigene Routine verbogen werden könnte.

Enthalten in der Tastaturabfrage ist auch die Überprüfung der RUN/STOP-Taste, die aber nur zusammen mit den in dem UDTIM-Aufruf voreingestellten Flaggen funktioniert. Deshalb wird das Abschalten der RUN/STOP-Taste im allgemeinen dadurch durchgeführt, daß man den IRQ-Vektor auf 59956 stellt und damit den ersten JSR-Befehl überspringt. Allerdings wird auf diese Weise auch die TI$-Uhr nicht weitergestellt.

60030	**LDA 56333**	Das ist das Unterbrechungs-Kontrollregister des IRQ-CIA, das hier durch Auslesen gelöscht wird.

Den Abschluß der IRQ-Routine bildet nun noch das Zurückschreiben der Register:

60033	**PLA**	Zurückholen des
	TAY	Y- und
	PLA	
	TAX	des X-Registers
	PLA	sowie des Akku.
60038	**RTI**	Damit kehrt der Computer zu dem durch den IRQ unterbrochenen Programm zurück.

Somit hätten wir's. Nun können wir je nach Bedarf entscheiden, welche von diesen Servicetätigkeiten wir bei einem eigenen IRQ-Programm brauchen: Die Uhr TI$, die Cursorbehandlung, die Abfrage der Recordertasten und die Tastaturabfrage.

Sehen wir uns nun an, was geschieht, wenn ein BRK-Kommando der Auslöser war.

57. BRK-Unterbrechung

Wir hatten vorhin am Scheideweg zwischen IRQ und BRK den letzteren links liegen gelassen. Normalerweise verwendet man beim Programmieren in Assembler ja ein Software-Instrument wie zum Beispiel den SMON, der so gebaut ist, daß der BRK-Vektor, welchen wir vorhin kennengelernt haben ($316/317 = 790/791) auf die Registeranzeige weist. Was geschieht eigentlich, wenn der BRK-Vektor unverändert bleibt, so also, wie er im Einschaltzustand des Computers vorliegt?

Dann zeigt er auf die Adresse 65126 ($FE66), wo ein Teil der NMI-Routine zu finden ist (Siehe auch das Flußdiagramm in Bild 42):

65126	**JSR 64789**	Sprung ins Programm RESTOR, in dem alle Vektoren (788-819) gemäß einer ROM-Liste auf ihre Ausgangswerte gesetzt werden.
	JSR 64931	Sprung in das Programm I/O-RESET.

In diesem Programm werden die beiden CIAs auf die Anfangswerte gestellt.

	JSR 58648	Sprung in ein Programm, welches zuerst den VIC-II-Chip initialisiert, dann einen Bildschirmeditor-RESET durchführt. Nach Beenden dieser Routine ist der Bildschirm gelöscht.
	JMP (40962)	

Mit diesem indirekten Sprung ist die BRK-Unterbrechung beendet. Man sieht aber jetzt schon deutlich, daß es sich hier nicht um eine Unterbrechung im eigentlichen Sinn handelt, vielmehr um einen Abbruch. In 40962/40963 steht die Adresse des Basic-Warmstarts (58235). Danach befindet sich der Computer im READY-Zustand in der Eingabe-Warteschleife.

Das Zurückholen der Register und ein RTI erübrigt sich hier, weil ohnehin viele Werte aus dem unterbrochenen Programm inzwischen weitgehend zerstört sind und alle Unterbrechungskontrollregister (CIAs und VIC-II-Chip) neu belegt wurden. Ein unkontrollierter BRK hat also fatale Folgen!

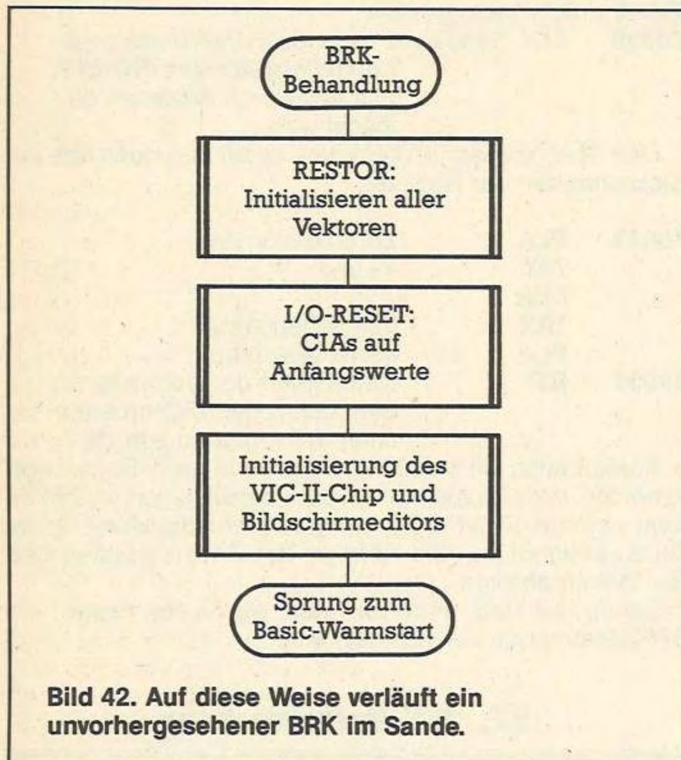


Bild 42. Auf diese Weise verläuft ein unvorhergesehener BRK im Sande.

Wenden wir uns nun der Firmware zu, die zur Bearbeitung eines NMI vorgesehen ist (dazu sehen Sie sich bitte in Bild 43 das Flußdiagramm an).

In der letzten Folge erfuhren wir, daß auch für diese Unterbrechung am Ende des Speichers ein Vektor vorhanden

58. Was macht ein NMI?

ist, nämlich $FFFA/FFFB (65530/65531). Dort steht die Adresse 65091 ($FE43), die nun in den Programmzähler gelangt und damit startet das folgende Programm:

```
65091   SEI          Unterbrechungen niedrigerer
                     Priorität werden gesperrt.
        JMP (792)
```

Das ist nun wieder ein für uns sehr interessanter Vektor 792/793 ($318/319), der — weil er im RAM-Bereich liegt — verstellbar ist. Genau das haben wir am Ende der letzten Folge getan mittels des M-Kommandos von SMON um den NMI zu testen, den wir mit der RESTORE-Taste ausgelöst haben. Der vorher eingestellte Wert in diesem Vektor ist die Adresse 65095 ($FE47), also direkt der nächste Befehl nach dem indirekten Sprungbefehl.

```
65095   PHA          Ebenso wie vorhin beim IRQ
        TXA          werden hier die Inhalte des Ak-
        PHA          ku und der
        TYA          Register auf den Stapel gescho-
        PHA          ben.
        LDA #127     das ist binär 01111111.
        STA 56589    Sperrt alle weiteren NMI-
                     Anforderungen
        LDY 56589    NMI-CIA Kontrollregister laden.
```

NMI
IRQ sperren
Register auf Stapel
CIA-NMI sperren
NMI-CIA-Kontrollregister laden
War CIA2 der NMI-Auslöser ? — J / N
Prüfung auf Modul
Modul vorhanden ? — J: Modulstart / N
Tastaturabfrage vorbereiten
STOP Abfrage der Stop-Taste
Nur Restore-Taste gedrückt ? — N: Programmabbruch / J
RS232C-Behandlung
Register vom Stapel holen
RTI

Bild 43. Flußdiagramm zum Ablauf einer NMI-Unterbrechung.

	BMI 65138	Wenn der NMI von der RESTORE-Taste kam ist Bit 7 des Registers =0, sonst =1 (bei NMI-Anforderung durch NMI-CIA). Wenn also nicht durch die RESTORE-Taste, erfolgt Sprung.

An dieser Stelle läuft nun das Programm weiter, wenn die RESTORE-Taste der NMI-Auslöser war:

65110	**JSR 64770**	Das ist ein Unterprogramm, welches prüft, ob ein Modul ab $8000 vorhanden ist.

Dies wird dadurch angezeigt, daß von $8004 bis $8008 die Werte stehen: 195, 194, 205, 56, 48 (das ist »CBM80«).

	BNE 65118	Wenn kein Modulprogramm ab $8000 vorliegt, erfolgt ein Sprung.
	JMP (32770)	Falls Modul.

Wenn ein Modul angezeigt wurde, erfolgt der indirekte Sprung nach den Vektor $8002/8003, der vom Modul vorgegeben wird. Das kann man auch nutzen, um eigene Maschinenprogramme durch einen Druck auf die RESTORE-Taste zu starten. Man muß dann nur in die Speicherstellen $8002 bis 8008 die geforderte Zieladresse beziehungsweise »CBM80« schreiben.

Der nun folgende Abschnitt wird nur angesprungen, wenn die RESTORE-Taste der NMI-Auslöser war:

65118	**JSR 63164**	Das ist ein Programmteil, der auch schon von der IRQ-Routine (nach dem Weiterstellen von TI$) durchlaufen wird. Hier werden einige Voreinstellungen für die Tastaturabfrage erledigt, die insbesonders die RUN/STOP-Taste betreffen.
	JSR 65505	Kernelroutine STOP.

Dort befindet sich ein indirekter Sprung über den Vektor 808/809 ($328/329), also auch ein verstellbarer RAM-Vektor. Im Normalfall zeigt dieser Vektor auf 63213 ($F6ED). Dort wird geprüft, ob die RUN/STOP-Taste gedrückt ist. Eine andere Methode zum Ausschalten des RUN/STOP bietet sich hier an, die die Uhr TI$ ungeschoren läßt.

	BNE 65138	Falls nur die RESTORE-Taste (also ohne RUN/STOP) gedrückt ist, erfolgt nun ein Sprung.

Waren aber sowohl die RUN/STOP- als auch die RESTORE-Taste gedrückt, dann folgt nun ein Programmabbruch, der uns schon von BRK her bekannt ist. Hier wie dort endet das Ganze dann mit dem Reset der I/O-Bausteine, des VIC-II-Chips, der Vektoren, des Bildschirmeditors und das Ergebnis ist ein Basic-Warmstart.

Ab 65138 befindet sich der Rest der NMI-Routine, an die das Programm springt, wenn

1) die NMI-Anforderung nicht von der RESTORE-Taste kommt oder

2) zwar von dieser Taste kommt, aber die RUN/STOP-Taste nicht gedrückt ist.

65138 bis 65211		Dieser ganze Abschnitt ist zur Behandlung der RS232C-Schnittstelle eingerichtet.
65212	**PLA**	Abschluß des NMI durch Rückschreiben des Akku und der Register vom Stapel
	TAY	
	PLA	
	TAX	
	PLA	
65217	**RTI**	Rückkehr zum unterbrochenen Programm.

Wenn Sie sich nun mal unser kleines Demo-Programm aus Kapitel 55 ansehen, dann werden Sie feststellen, daß der Programmteil bis $600E lediglich den ersten Teil der normalen NMI-Routine kopiert. Die Prüfung auf das Modul und die RUN/STOP-Taste werden übersprungen. Statt dessen erfolgt nach der Abarbeitung des für die RESTORE-Taste gebauten Programmes das Ende der NMI-Routine ($FEBC = 65212). Im anderen Fall, wenn also die RESTORE-Taste nicht der Auslöser des NMI war, wird in die normale Routine ab 65138 eingemündet.

59. Eigentlich keine Unterbrechung: RESET

Weil wir alle Unterbrechungen hier bearbeiten wollen, soll auch der RESET angesprochen werden. Es handelt sich dabei aber nicht um eine Unterbrechung im bisher definierten Sinn. Mir fällt allerdings kein Platz ein, wo der RESET besser hinpassen würde. Ähnlich wie bei NMI und IRQ wird auch hier ein Vektorinhalt in den Programmzähler geladen, der in den höchsten Speicheradressen zu finden ist (Auch hierzu wieder ein Flußdiagramm in Bild 44).

Dieser Vektor liegt in $FFFC/FFFD. Der Inhalt ist die Adresse 64738 ($FCE2) und genau dort geht das Programm dann weiter:

64738	**LDX #255**	Im ersten Teil wird der Stapelspeicher initialisiert.
	SEI	Verhindern von IRQ
	TXS	Stapelzeiger auf $FF
	CLD	Dezimal-Modus ausschalten (falls er eingeschaltet war).
	JSR 64770	Das ist wieder das Unterprogramm, das auf ein Modul prüft.

Hier ergibt sich die Möglichkeit, auch beim RESET einzu-

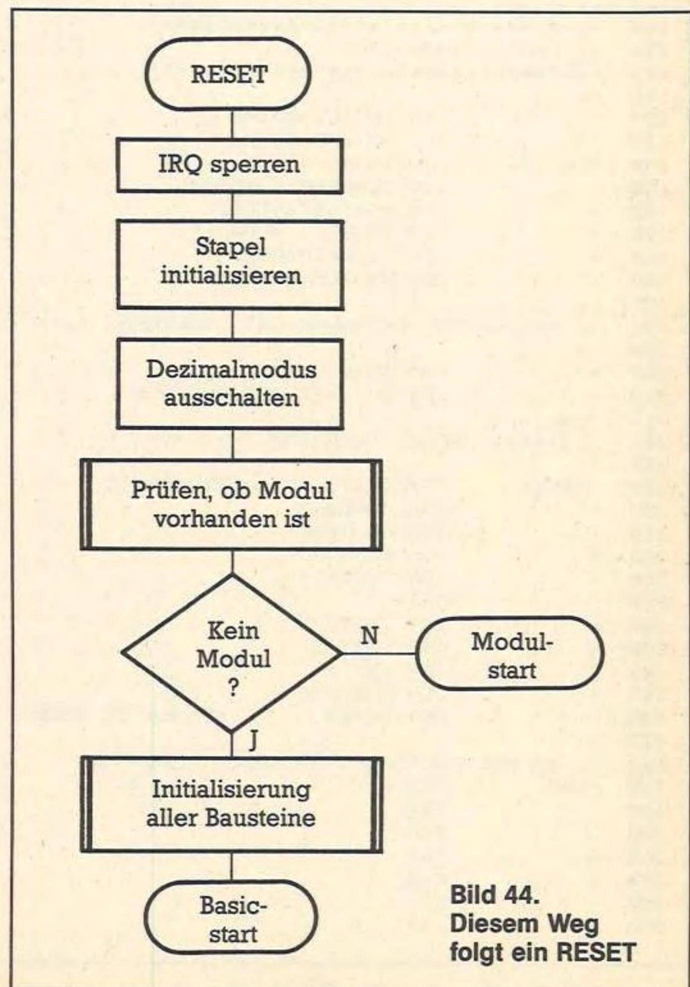


Bild 44. Diesem Weg folgt ein RESET

greifen, indem man die Kennung »CBM80« an die abgefragten Speicherstellen ($8004 bis $8008) legt.

BNE 64751 Falls kein Modul, erfolgt Sprung.

64748 JMP(32768)

Dieser indirekte Sprung erfolgt nach dem Vektorinhalt von $8000/8001 = 32768/32769. Das ist ein anderer Vektor als wir ihn vorhin beim NMI hatten (dort war es $8002/8003 = 32770/32771). So kann ein anderer Programmteil angesteuert werden als durch den NMI, was übrigens auch dringend erforderlich ist, weil der Stapelzeiger zerstört wurde.

64751 Hier läuft das Programm weiter, falls keine Modulkennung erkannt wurde.

Der ganze Rest dient dem Versetzen des Computers in den Einschaltzustand. Allerdings bin ich davon überzeugt, daß noch irgendein Unterschied bestehen muß zwischen dem einfachen Aus- und Wiederanschalten des Computers und einem RESET. Es hat sich nämlich bei einigen Programmen gezeigt, daß sie nach einem RESET fehlerhafte Verläufe nehmen können, was nach einem totalen Aus- und Wiederanschalten nicht zu beobachten war. Der Grund für diesen Unterschied liegt (für mich) noch im dunkeln.

60. Die Sache mit dem Modulstart

Sowohl beim RESET als auch beim NMI haben wir festgestellt, daß der Modulstart-Bereich ab $8000 eine besondere Rolle spielt. In Bild 45 finden Sie noch mal zusammengefaßt, was sich dort findet, wenn ein Modul vorhanden ist.

```
100  -          .LI 1,3
110  -          .BA $8000
112  -;
114  -;********************************
116  -;  MODULSIMULATION
117  -;********************************
118  -;
120  -          .EQ INITCZ=$E3BF
130  -          .EQ INITMS=$E422
140  -          .EQ INITV=$E453
150  -          .EQ SCREENCLR=$E544
160  -          .EQ RAMTEST=$FD50
170  -          .EQ IORESET=$FDA3
180  -          .EQ TVTAKT=$FF5B
190  -          .EQ RESTOR=$FF8A
192  -;
194  -; ****** MODULKENNUNG UND -VEKTOREN ****
196  -;
200  -          .WO RESET,NMI
210  -          .BY $C3,$C2,$CD,$38,$30
212  -;
214  -;******  RESET-PROGRAMM ********
216  -;
220  -RESET     STX $D016      ;RESET-BIT
230  -          JSR IORESET
240  -          JSR RAMTEST
250  -          JSR RESTOR
260  -          JSR TVTAKT
270  -          CLI
280  -          JSR INITV
290  -          JSR INITCZ
300  -          JSR INITMS
310  -          JSR SCREENCLR
320  -          JMP $C000      ;SPRUNG IN SMON
322  -;
324  -; ** NMI-PROGRAMM (RESTORE-TASTE) **
330  -NMI       PLA
340  -          TAY
350  -          PLA
360  -          TAX
370  -          PLA
380  -          RTI
390  -          .SY 1,4
```

Listing 6. Simulation eines Moduls

Speicherplatz ($)	8000	8001	8002	8003	8004	8005	8006	8007	8008
Inhalt	LSB	MSB	LSB	MSB	C	B	M	8	0
	RESET-Vektor		NMI-Vektor						

Bild 45. Diesen Inhalt müssen die Speicherstellen $8000 bis $8008 haben, damit ein Modulstart stattfindet.

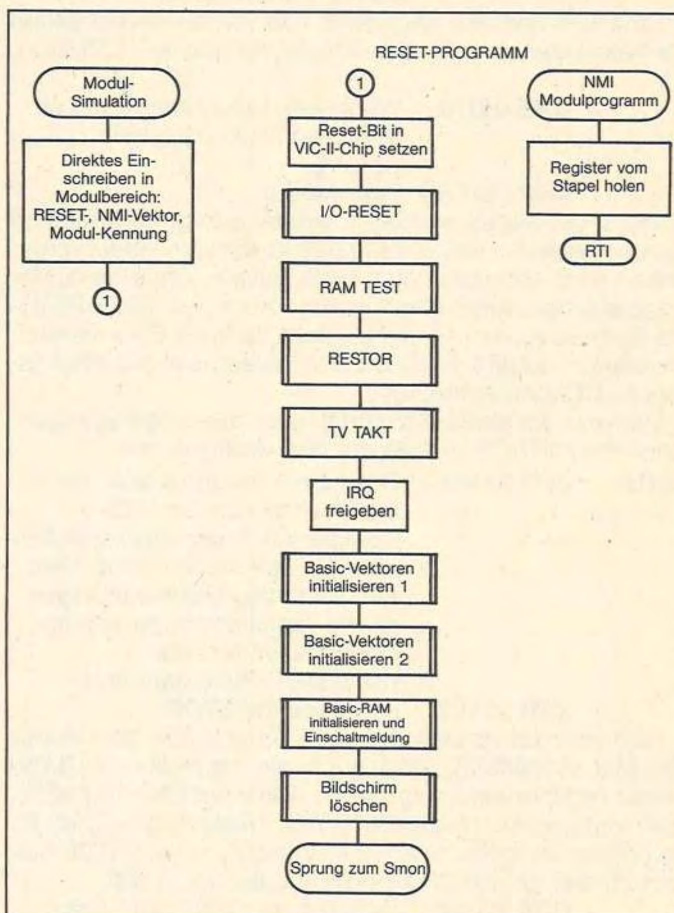


Bild 46. Das Flußdiagramm zum Listing 6, Modulsimulation

Wir wollen im folgenden Beispielprogramm (Listing 6) ein Modul simulieren, indem wir den SMON mittels des RESET anspringen. Der NMI — also die RUN/STOP-RESTORE-Tastenkombination — soll dabei wirkungslos gemacht werden.

Bild 46 zeigt ein Flußdiagramm dieses Beispielprogrammes:

Achten Sie bitte darauf, daß Sie nach dem Eintippen des Programmes speichern und — natürlich — daß die SMON-Version ab $C000 im Speicher vorliegt. Wenn Sie nun mal die RESTORE-Taste — oder RUN/STOP und RESTORE — drücken, passiert offensichtlich nichts. Das liegt daran, daß unser Programm lediglich die auf den Stapel gelegten Register wieder zurückholt und aus der Unterbrechung mit RTI ins normale Geschehen zurückkehrt.

Haben Sie einen Reset-Taster eingebaut? Dann drücken Sie doch mal drauf. Zunächst erkennen Sie den normalen Reset-Verlauf. Dann meldet sich aber nicht wie gewohnt die Nachricht CBM-Basic..., sondern der SMON mit einer Registeranzeige. Das RESET-Programm ab $602E folgt dem Firmware-Programm. Auf diese Weise (und mittels eines Autostart) sichern sich Softwarehäuser manchmal gegen unbefugtes Kopieren ihrer Programme.

Sowohl was die Hardware als auch die Firmware für die Unterbrechungsbehandlung angeht, haben wir nun einen guten Überblick gewonnen. Es ist jetzt an der Zeit, daß wir uns

61. Nutzung der Unterbrechungen

ansehen, auf welche Weise man dieses Reservoir an vielfältigen Möglichkeiten für sich nutzen kann. Dazu soll uns ein Überblick dienen:

1) Auslösung der Unterbrechung durch Hardware-Einwirkungen.

Da hätten wir beispielsweise den Userport oder den Expansion-Port, über die wir per CIAs Unterbrechungen anfordern können. Um es gleich zu sagen: Damit werden wir uns nicht auseinandersetzen. Meine Kenntnisse auf diesem Gebiet sind zu dünn. Aber vielleicht verstehen Sie das auch mal als Aufforderung, Ihre Versuche dazu anderen zu offenbaren? Also: Schreiben Sie doch mal!

2) Unterbrechungsauslösung per Software:

Damit haben wir immer noch ein weites Feld von Möglichkeiten vor uns:

2a) Vorgesehene Nutzungen des IRQ

— mittels des VIC-II-Chips.

Da können wir uns auf den Rasterzeileninterrupt, die Sprite/Hintergrund- oder die Sprite/Sprite-Kollision stützen.

— oder mit Hilfe des CIA1

Da ist es vor allem der 60mal pro Sekunde auftretende Timer A-Unterlauf, der uns interessieren soll.

2b) Vorgesehene Nutzungen des NMI

— CIA2: Läßt man die RS232C-Schnittstellenbehandlung außer acht, dann gibt es keine vorgesehene Nutzung.

— RESTORE: Zusammen mit der RUN/STOP-Taste kann man die vorgegebene Routine verändern, wie wir es schon in einigen Beispielen gezeigt haben.

Wir können außerdem noch unterscheiden zwischen Nutzungen, die periodisch stattfinden sollen (zum Beispiel eine spezielle Tastaturabfrage) und solchen, die stochastisch (= zufallsabhängig) oder willkürlich erfolgen (zum Beispiel Drücken der RESTORE-Taste). Beides ist auch durchführbar bei:

2c) Nicht vorgesehene Nutzung der Unterbrechungen.

Da bietet sich vor allem der meistens völlig brachliegende CIA2 an mit seinen beiden Timern und der Alarmfunktion.

Wenn Sie aber erst einmal vertraut sind mit der Unterbrechungs-Programmierung und auch etwas Zeit zum Tüfteln investieren, finden Sie bestimmt noch eine ganze Menge weiterer Möglichkeiten.

Bei mehreren gleichartigen Unterbrechungsanforderungen (zum Beispiel IRQs) muß noch ein Weg gefunden werden, wie zwischen den dann vielleicht anfallenden unterschiedlichen Service-Routinen differenziert werden kann. Denkbar wären beispielsweise Aufgabenstellungen wie:

Jeder dritte Timer-IRQ soll den Joystick abfragen, oder RESTORE+h soll den Hilfsbildschirm zeigen, RESTORE+z soll den aktuellen Bildschirm wieder restaurieren, etc.

Sie sehen, eine große Menge Arbeit wartet auf uns. Nicht zu allen Möglichkeiten werde ich hier Beispielprogramme zeigen. Außerdem dürfen die dann auch nicht zu undurchsichtig sein und man sollte möglichst den Erfolg eines solchen Demo-Programmes auf dem Bildschirm erkennen können. Trotzdem hoffe ich, daß die nachfolgend und in den nächsten Kapiteln gezeigten Programmlösungen ausreichen, Ihnen die Unterbrechungs-Behandlung mit eigenen Routinen durchschaubar zu machen. Ich will Ihnen aber nicht verschweigen, daß auch mir noch längst nicht alle Geheimnisse der Unterbrechungsprogrammierung offenbar geworden sind. Oft finde ich mich unversehens in Programm-Sackgassen wieder. Das soll Ihnen als kleiner Trost dienen, wenn Sie mal nach dem 1001. Absturz müde und mit rauchendem Kopf vor Ihrem »Commodore-Ungeheuer sitzen«.

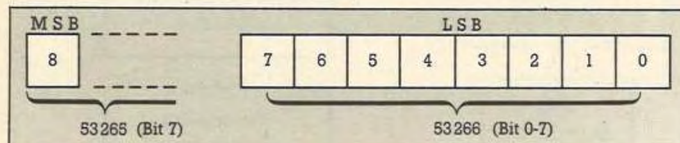


Bild 47. So sieht das 9-Bit-Register im VIC-II-Chip aus, welches die Rasterzeilen mitzählt.

```
PROGRAMM 7

,6000  78         SEI
,6001  A9 28      LDA #28
,6003  8D 14 03   STA 0314
,6006  A9 60      LDA #60
,6008  8D 15 03   STA 0315
,600B  A9 F8      LDA #F8
,600D  8D 12 D0   STA D012
,6010  AD 11 D0   LDA D011
,6013  29 7F      AND #7F
,6015  8D 11 D0   STA D011
,6018  A9 81      LDA #81
,601A  8D 1A D0   STA D01A
,601D  A9 00      LDA #00
,601F  8D 20 D0   STA D020
,6022  A9 04      LDA #04
,6024  85 02      STA   02
,6026  58         CLI
,6027  60         RTS
--------------------------
,6028  AD 19 D0   LDA D019
,602B  8D 19 D0   STA D019
,602E  30 07      BMI 6037
,6030  AD 0D DC   LDA DC0D
,6033  58         CLI
,6034  4C 31 EA   JMP EA31
--------------------------
,6037  AD 12 D0   LDA D012
,603A  C9 F8      CMP #F8
,603C  B0 11      BCS 604F
,603E  18         CLC
,603F  65 02      ADC   02
,6041  8D 12 D0   STA D012
,6044  A0 03      LDY #03
,6046  88         DEY
,6047  D0 FD      BNE 6046
,6049  EE 20 D0   INC D020
,604C  4C 81 EA   JMP EA81
--------------------------
,604F  A9 00      LDA #00
,6051  8D 20 D0   STA D020
,6054  A9 32      LDA #32
,6056  8D 12 D0   STA D012
,6059  4C 81 EA   JMP EA81
--------------------------
,605C  78         SEI
,605D  A9 00      LDA #00
,605F  8D 1A D0   STA D01A
,6062  A9 31      LDA #31
,6064  8D 14 03   STA 0314
,6067  A9 EA      LDA #EA
,6069  8D 15 03   STA 0315
,606C  A9 0E      LDA #0E
,606E  8D 20 D0   STA D020
,6071  58         CLI
,6072  60         RTS
--------------------------
```

Listing 7. Das im Artikel entwickelte Programm auf einen Blick.

62. Ein Programm zum VIC-II-IRQ

Sehr schöne Effekte lassen sich durch eine periodische IRQ-Anforderung per Rasterzeileninterrupt mittels des VIC-II-Chip erzielen. Deshalb ist sowas auch ein beliebtes Objekt für Demos von Unterbrechungsprogrammen. Als Ziel setzen wir uns, einen Bildschirm zu konstruieren, dessen Rahmen in allen Farben schillert.

Leser des Buches »C 64: Wunderland der Grafik« werden diese Möglichkeit des VIC-II-Chip schon kennen: Man kann dem Kathodenstrahl, der über den Monitor huscht, um das Bild zu erzeugen, über zwei Register folgen, die Rasterregister, wo jede Rasterzeile mitgezählt wird. Ohne an dieser Stelle allzusehr in die Einzelheiten einzugehen, soll hier nur bemerkt werden, daß die Numerierung dabei etwa von 0 bis 280 geht, weil auch der Rahmen und nicht sichtbare Teile des Bildschirmes vom Strahl überstrichen werden. Wo das Textfeld anfängt, ist von Monitor zu Monitor (oder Fernseher) etwas unterschiedlich. Bei mir beginnt es oben in Rasterzeile 50 und endet unten bei Zeile 248. Sollten die im Beispielprogramm 7 (Listing 7) voreingestellten Randwerte bei Ihnen also anders sein, können Sie sie durch einige später noch angegebenen POKEs ändern. Die beiden Rasterzeilenregister sind:

$D012 (53266)
$D011 (53265)

Von $D011 allerdings ist nur das Bit 7 als msb der Rasterzeilenzahl für uns von Bedeutung. Bild 47 soll diese Belegung deutlich machen:

Das Interessante an diesen Registern ist nun, daß man auch in sie schreiben kann. Die auf diese Weise festgelegte Rasterzeile ist dann der Auslöser des IRQ, falls dieser im Interrupt-Enable-Register $D01A freigegeben wurde (das kennen wir noch aus Kapitel 51).

Damit kann also unsere primäre Unterbrechungsquelle (der VIC-II-Chip) programmiert werden. Halten wir die zwei Schritte dazu noch mal fest:

1) Rasterzeile festlegen, bei der ein IRQ ausgelöst werden soll, durch Einschreiben in die Register $D012 und Bit 7 von $D011.

2) Freigeben des Rasterzeileninterrupts durch Einschreiben von 1000 0001 in das Interrupt-enable-Register $D01A.

Der nächste Schritt betrifft die Bearbeitung des IRQ durch die CPU. Wie wir vorhin sahen, springt das Programm beim IRQ mittels eines indirekten Sprunges, der auf den Vektor 788/9 ($314/5) zugreift. Dieser Vektor muß nun auf die eigene Routine verbogen werden, also:

3) Vektor $314/5 auf die IRQ-Service-Routine richten.

Damit wären alle Vorbereitungen getroffen. Der Rest liegt nun ganz bei uns — beziehungsweise bei dem von uns zu schreibenden Service-Programm. In Bild 48 finden Sie ein Flußdiagramm unseres Beispielprogrammes 7.

Gehen wir nun an die Realisierung. Zunächst also die Initialisierung, die wir bei $6000 (also durch SYS 24576 zu starten) beginnen lassen:

```
6000   SEI          Sperren von IRQs
Schritt 3:
6001   LDA #$28     LSB der IRQ-Routine
6003   STA $0314    in IRQ-Vektor-LSB
6006   LDA #$60     MSB der IRQ-Routine
6008   STA $0315    in IRQ-Vektor-MSB
Schritt 1:
600B   LDA #$F8     Rasterzeile, bei der das Textfen-
                    ster endet. Von da an soll der
                    Rahmen schwarz sein.
600D   STA $D012    in Rasterzeilen-Register (LSB)
                    schreiben.
```

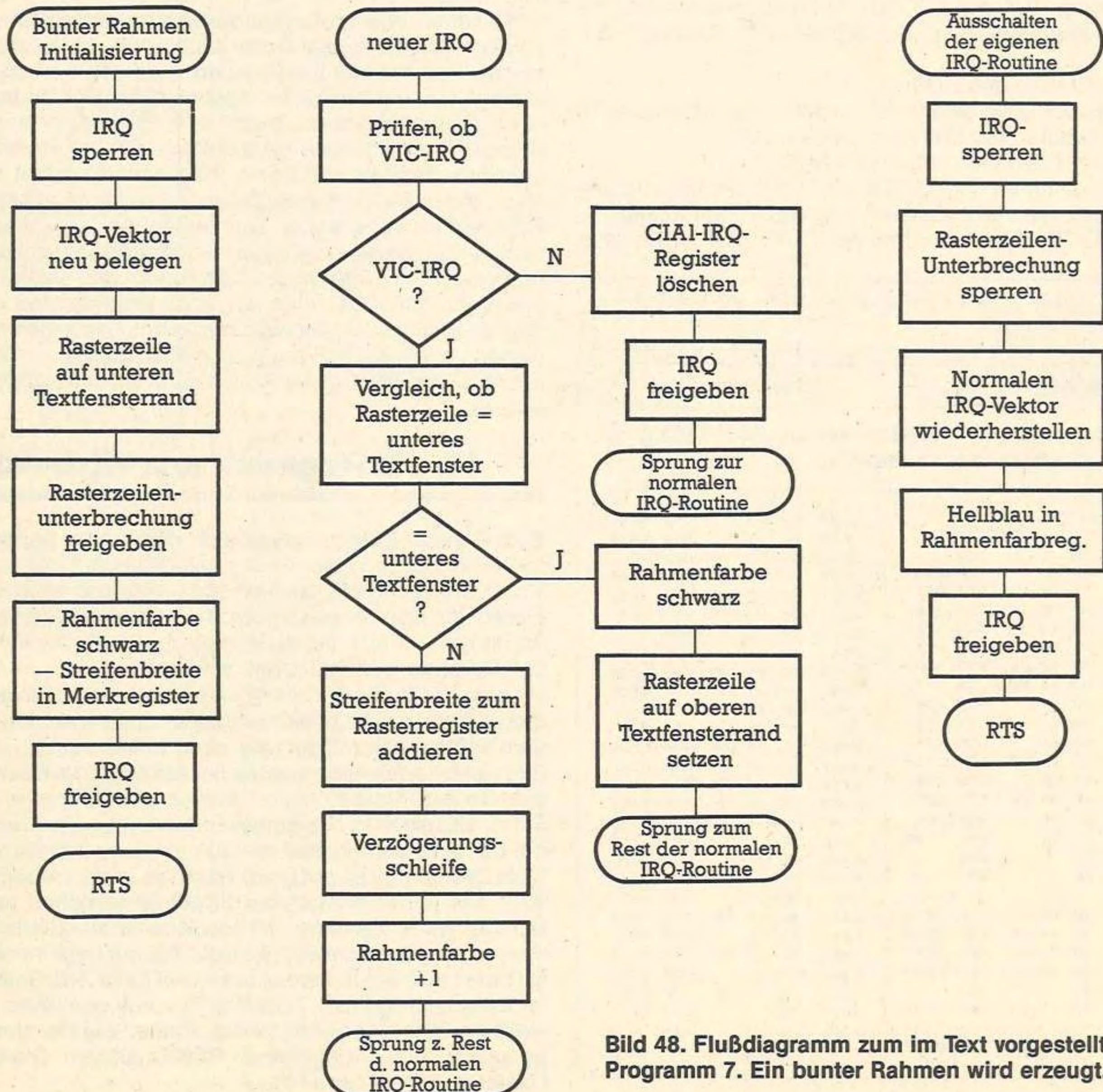


Bild 48. Flußdiagramm zum im Text vorgestellten Programm 7. Ein bunter Rahmen wird erzeugt.

6010	LDA $D011	Register mit dem msb des Rasterzeilenzählers
6013	AND #$7F	0111 1111 löscht das Bit7
6015	STA $D011	Zurückschreiben. Damit ist die Rasterzeile, die den IRQ auslösen soll, festgelegt.

Schritt 2:

6018	LDA #$81	1000 0001 wird nun
601A	STA $D01A	ins IRQ-enable-Register geschrieben, um den Rasterzeilen-IRQ zuzulassen.

Festlegen einiger Startwerte:

601D	LDA #$00	Farbe schwarz
601F	STA $D020	in Rahmen schreiben
6022	LDA #$04	Streifenbreite in
6024	STA $02	Merkregister schreiben.
6026	CLI	IRQ freigeben
6027	RTS	Ende der Initialisierung.

Von nun an laufen alle IRQs über unsere eigene Routine, die bei $6028 beginnt.

Zunächst müssen wir prüfen, ob die Unterbrechung vom VIC-II-Chip kommt oder vom CIA1:

6028	LDA $D019	IRQ-Request-Register des VIC-II-Chip (siehe Kapitel 51). Dort ist Bit 7 gesetzt, wenn die Anforderung vom VIC-II-Chip kam.
602B	STA $D019	Zurückschreiben
602E	BMI $6037	Sprung, falls VIC-IRQ, sonst CIA-IRQ.

Bearbeiten eines CIA-IRQ:

6030	LDA $DC0D	Löschen des CIA1 Unterbrechungs-Kontrollregisters.
6033	CLI	IRQ zulassen. Damit können innerhalb eines CIA- IRQ auch unsere VIC-IRQs geschehen.
6034	JMP $EA31	Bearbeitung des CIA-IRQ durch die normale Routine.

Unser Programm für VIC-II-IRQs:

6037	LDA $D012	Rasterzeilen-Register laden um festzustellen, welche Zeile den IRQ auslöste.
603A	CMP #$F8	Vergleich mit Ende des Textfensters.
603C	BCS $604F	Wenn unterhalb des Textfensters, Sprung.

Der folgende Programmteil ist wirksam, wenn der IRQ-Auslöser eine Zeile in Höhe des Textfensters war:

603E	CLC	Addition vorbereiten.
603F	ADC $02	Streifenbreite aus dem Merkregister addieren.
6041	STA $D012	Neuen Wert in Rasterzeilen-Register schreiben.

Damit wird eine neue Rasterzeile als IRQ-Auslöser festgelegt, die um die Streifenbreite tiefer liegt als die vorhergegangene.

Es folgt eine kleine Verzögerungsschleife, die aber nur zum Experimentieren eingebaut wurde:

6044	LDY #$03	Schleifen-Startwert
6046	DEY	Herunterzählen
6047	BNE $6046	NEXT Y, bis Y=0.

Ändern der Rahmenfarbe bis zum nächsten Raster-IRQ:

6049	INC $D020	Farbcode+1. Wenn Code im Rahmenfarbregister größer als 15 wird, fängt wieder Farbcode 0 an, weil die Bits 5-7 keine Funktion haben.

Abschließend erfolgt der Rücksprung in den Rest der normalen IRQ-Routine:

604C	JMP $EA81	Siehe unsere Untersuchung der IRQ-Firmware.

Damit ist der Rahmen in Höhe des Textfensters behandelt. Es schließt sich nun der Teil an, der die Rahmenbereiche unter- und oberhalb bearbeitet:

604F	LDA #$00	Farbcode schwarz
6051	STA $D020	in Rahmenfarb-Register.
6054	LDA #$32	Rasterzeile, bei der oben das Textfenster beginnt.
6056	STA $D012	In Rasterzeilen-Register schreiben
6059	JMP $EA81	Abschluß durch Sprung zum Ende der normalen IRQ- Routine.

Damit ist festgelegt, daß ober- und unterhalb des Textfensters die Rahmenfarbe schwarz wird.

Unsere eigene Routine ist jetzt abgeschlossen. Zum guten Ton gehört es, dem Benutzer auch die Möglichkeit zu öffnen, diese Routine wieder abzuschalten. Das erfolgt im letzten Programmteil, der mittels SYS24688 aktiviert werden kann:

605C	SEI	IRQ sperren
605D	LDA #$00	Raster-IRQ
605F	STA $D01A	abschalten
6062	LDA #$31	IRQ-Vektor
6064	STA $0314	restaurieren
6067	LDA #$EA	auf den
6069	STA $0315	Normalwert.
606C	LDA #$0E	Farbcode hellblau
606E	STA $D020	in Rahmenfarb-Register schreiben
6071	CLI	IRQ zulassen
6072	RTS	

Unser Programm ist komplett. Speichern Sie es bitte vor dem Starten ab. Nach dem SYS 24576 finden Sie einen hübschen bunten Rahmen vor, oberhalb und unterhalb des Textfensters ist er schwarz. Besonders gut — finde ich — sieht das Ganze aus, wenn man die Hintergrundfarbe des Textfensters auch auf Schwarz setzt (POKE 53281,0). Das Programm erlaubt noch einige Experimente:

Durch POKE-Kommandos in die Speicherstelle 2 kann die aktuelle Streifenbreite variiert werden, durch POKEs in die Zelle 24645 der Startwert der Verzögerungsschleife. Probieren Sie's doch mal aus. Eine Erkenntnis werden Sie gewinnen: In der Unterbrechungs-Programmierung spielt die Zeit eine wichtige Rolle. Das zeigt sich auch, wenn man zum Beispiel Cursorbewegungen durchführt: Die Streifen fangen an zu wandern.

Weitere Möglichkeiten zum Experimentieren sind gegeben, wenn Sie die Rasterzeilen verändern, die den oberen und unteren Rand des Textfensters markieren:

Durch POKE 24661,Zahl verschieben Sie die obere, durch POKE 24635,X:POKE 24588,X die untere Rasterzeile, von der an alles schwarz ist. Wie schon vorhin erwähnt, habe ich im Programm diese Werte auf 50 beziehungsweise 248 fixiert, weil genau dort auf meinem Monitor das Textfenster liegt.

Mit diesem Beispiel sollte es Ihnen nun möglich sein, auch andere Unterbrechungsprogramme zu schreiben, die sich der Rasterzeilen-Unterbrechung per VIC-II-Chip bedienen. Eine Bemerkung sollte ich Ihnen noch auf den Weg Ihrer eigenen Versuche mitgeben: Der Elektronenstrahl, der über den Bildschirm saust und beim Erreichen des von uns bestimmten Rasterzeilenwertes zum Auslösen des IRQ führt, ist enorm schnell. Die Service-Programme dürfen deshalb nicht zu lang sein, sonst steht der nächste IRQ schon wieder an, bevor der vorangegangene bearbeitet ist.

Lassen Sie uns kurz rekapitulieren: Als primäre Unterbrechungsanforderer hatten wir drei Bausteine unseres Computers benannt, nämlich den VIC-II-Chip und die bei-

den CIA-Bausteine. CIA kommt von »Complex Interface Adapter« und ist die Bezeichnung für die beiden Ein- und Ausgabe-Bausteine, die den gesamten Verkehr zwischen dem zentralen Gehirn unseres C 64 und der Peripherie managen. Wir hatten bemerkt, daß ein CIA, der IRQ-CIA (Adressen von 56320 bis 56575), ausschließlich für die maskierbaren Unterbrechungen zuständig ist. Dazu gehören die 60mal pro Sekunde stattfindenden »Timer-Interrupts«, die die Cursor-Behandlung, die TI$-Uhr, die Tastaturabfrage etc. bearbeiten. Der andere CIA, genannt NMI-CIA, (Adressenraum 56576-56831) ist nur für die nicht maskierbaren Unterbrechungen verantwortlich und wird bei normaler Nutzung des C 64 so gut wie nie eingesetzt. Ich gehe im folgenden davon aus, daß Sie die RS232C-Schnittstelle in Ihren Computer nicht einsetzen. Sollte das aber der Fall sein, dann müßten Sie darauf achten, die folgenden Beispiele - die den NMI-CIA betreffen - ohne gleichzeitigen Betrieb dieser Schnittstelle anzuwenden, weil sich sonst Störungen ergeben könnten.

63. Unterbrechungen mit den CIAs

In Kapitel 52 haben wir uns ein Register (das Register 13, Interrupt-Kontrollregister) der CIAs schon genauer angesehen und auch die Unterschiede beider Bausteine festgestellt. Dort war dann die Rede von Timern, Echtzeituhren, Alarm-Funktionen etc. Was es damit auf sich hat und wie man diese Möglichkeiten nutzen kann, das soll nun unser Thema sein. Wir werden uns dazu alle Register der CIAs genauer ansehen, die für die von uns ausgewählten Unterbrechungsoptionen eine Rolle spielen. Dabei fallen einige unter den Tisch - das habe ich aber schon in Kapitel 52 angekündigt —, nämlich diejenigen, die mit dem Verkehr über den seriellen Port, beziehungsweise über die RS232C-Schnittstelle, zu tun haben. Es bleibt dann anderen - kompetenteren - überlassen, darüber zu schreiben. Wie wäre es zum Beispiel mit Ihnen?

Auch so bleibt uns genug zu tun. In Tabelle 21 finden Sie zunächst eine Übersicht der von uns behandelten Register.

Sie sehen darin, daß jeder CIA über zwei sogenannte Timer (A und B) verfügt, sodann über die »Time of Day« (zu deutsch etwa »Tageszeit«) genannte Echtzeituhr mit vier Registern und schließlich noch über drei Kontrollregister, zu denen auch das schon erwähnte Register 13 gehört. Sehen wir uns zunächst die Timer an.

64. Die Timer der CIAs

Insgesamt verfügen wir über vier dieser Timer: Timer A und B im CIA1 und dasselbe noch mal im CIA2. Es handelt sich dabei um 16-Bit-Register, in die ein Startwert geschrieben werden kann, von dem an dann heruntergezählt wird. Jedesmal, wenn dann der Wert 0 unterschritten ist, gibt es für uns die Möglichkeit, bestimmte Ereignisse stattfinden zu lassen. Man kann diese Register unabhängig voneinander, aber auch kombiniert, benutzen. Ein Lesen des Registers liefert immer den momentan gerade aktuellen Wert. Ein Schreiben in das Register führt automatisch zum Festlegen eines Startwertes. Was an Optionen mit diesen Timern möglich ist, wird über Kontrollregister gesteuert. Das CRA (Register $ 0E) bezieht sich vor allem auf den Timer A, das CRB (Register $ 0F) auf Timer B. Die 16-Bit-Register werden - wie gewohnt - in der Form LSB/MSB betrieben. In den Timer A des CIA1 wird bei jedem I/O-Reset folgendes Wertepaar eingetragen:

56324	dezimal 37	LSB
56325	dezimal 64	MSB

Register Nr. ($)	Adresse (dez.) CIA-1	CIA-2	Name	Funktion	
04	56324	56580	TALO	TIMER A	LSB
05	56325	56581	TAHI	TIMER A	MSB
06	56326	56582	TBLO	TIMER B	LSB
07	56327	56583	TBHI	TIMER B	MSB
08	56328	56584	TOD10THS	1/10-Sekunden-Register	
09	56329	56585	TODSEC	Sekunden-Register	
0A	56330	56586	TODMIN	Minuten-Register	
0B	56331	56587	TODHR	Stunden-Register, AM/PM-Flagge	
0D	56333	56589	JCR	Unterbrechungs-Kontrollregister	
0E	56334	56590	CRA	Kontrollregister A	
0F	56335	56591	CRB	Kontrollregister B	

Tabelle 21. Die wichtigen Register der beiden CIAs.

Das entspricht einem Startwert von 16421. Im PAL-System hat der Quarz, der die Taktfrequenz bestimmt, eine Frequenz von 17.734472 MHz. Die Prozessorfrequenz errechnet sich daraus mittels Division durch 18 zu 985248.4 Hz (also etwas weniger als 1 MHz, was den europäischen C 64 langsamer macht als den amerikanischen, der etwas mehr als 1 MHz verwendet). Wenn mit dieser Geschwindigkeit der Timer heruntergezählt wird, erhält man genau einen Unterlauf alle 1/60 Sekunden. Das ist der Weg, eine kontrollierte Zeitspanne durch den Timer zählen zu lassen. Sei X der gesuchte Startwert, der zu einer Spanne von T Sekunden führt, dann kann man X berechnen mittels:

$$X = 985248.4 * T$$

Der Integerwert von X ist dann in ein LSB und ein MSB zu teilen und in die Timer-Register einzutragen. Allerdings ergibt sich so eine natürliche Grenze. Die höchste durch 2 Byte darstellbare Zahl ist ja 65535. Wenn wir diesen Wert in den Timer schreiben, dann ist er alle 1/15 Sekunden auf 0 heruntergezählt. Für längere Zeiten ist aber vorgesorgt. Die beiden Timer A und B sind kombinierbar (wie, dazu kommen wir gleich noch) zu einem 32-Bit-Register. Die höchste Zahl X ist dann:

$$4\,294\,967\,296 = 2^{32}$$

Damit kann im Extremfall eine Herabzählzeit von 1 Stunde, 12 Minuten und zirka 40 Sekunden eingeplant werden, was für die meisten Zwecke ausreichen dürfte.

Möchten Sie also genau eine Sekunde Spielraum haben beim Herunterzählen, dann muß die Zahl 985248 als 4-Byte-Integer-Wert in die Speicher von Timer A und Timer B gebracht werden. Das führt dann zu den Werten 0, 15, 8, 160 (weil 985248 = 0*16777216 + 15*65536 + 8*256 + 160). 0 und 15 gelangen als MSB beziehungsweise LSB in Timer B (also Register 07 und 06), 8 und 160 sind MSB und LSB für den Timer A (Register 05 und 04). Sehen wir uns nun an, wie wir dem Computer sagen, was mit diesen Startwerten in den Timer-Registern geschehen soll. Die beiden Kontrollregister CRA und CRB beziehen sich weitgehend auf die gleichnamigen Timer. Im Bild 49 finden Sie das Register $0E, also CRA und in Bild 50 das andere Kontrollregister CRB ($0F):

Die Bedeutung der Bits 0 bis 4 ist - jeweils für den dazugehörigen Timer - identisch:

Bit 0 — »0« an dieser Stelle führt zum sofortigen Anhalten des Timers. 1 in diesem Bit startet das Herunterzählen.

Bits 1 und 2 — Diese beiden Bits hängen mit dem externen Signalverkehr zusammen und sollen für uns außer acht bleiben.

Bit 3 — Ist dieses Bit = 1, dann ist der sogenannte »One Shot«-Betrieb des Timers aktiv. Das bedeutet, daß vom Startwert an heruntergezählt wird bis auf Null. Es findet nun das program-

mierte Ereignis statt (zum Beispiel ein IRQ). Anschließend wird der Startwert wieder eingeladen und der Timer gestoppt.

Im Gegensatz dazu läuft der »Continuous«-Betrieb, wenn das Bit den Wert 0 enthält. Dabei geschieht zunächst dasselbe wie beim One Shot Modus, der Timer wird aber nicht angehalten, sondern der ganze Vorgang wiederholt sich in einer Endlosschleife.

Bit 4 Ein Hineinschreiben einer 1 in dieses Bit erzeugt ein sofortiges Neuladen der Timer-Register mit dem Startwert. Dabei ist es gleichgültig, ob der Timer gerade läuft oder nicht. Schreibt man eine Null ein, hat das keine Wirkung.

Beim Lesen des Registers ist dieses Bit immer 0.

Zu diesem Bit und seiner Wirkung ist noch etwas zu sagen. Das Neuladen des Timers geschieht

- immer dann, wenn ein Unterlauf stattgefunden hat oder
- falls der Timer steht und in die Register ein Startwert geschrieben wird. Dabei ist der CIA so konstruiert, daß man kein zwangsweises Laden (also mit Bit 4 = 1) braucht, wenn man den Startwert in der Reihenfolge LSB, MSB in die Register bringt.

Die Bits 5 bis 7 haben nun unterschiedliche Bedeutung im CRA und im CRB:

Register CRA ($0E)

Bit 5: Ist dieses Bit gleich Null, dann wird im Systemtakt gezählt. Den hatten wir vorhin zur Zeitberechnung schon verwendet. Wenn das Bit auf 1 gesetzt ist, zählt der Timer externe Signale.

Bit 6: Spielt für den Signalverkehr über den seriellen Port eine Rolle und soll uns hier nicht weiter beschäftigen.

Bit 7: Damit steuert man nicht den Timer A, sondern dieses Bit bezieht sich auf die gleich noch zu behandelnde Echtzeituhr.

Register CRB ($0F)

Die Bits 5 und 6 sind hier im Zusammenhang von Bedeutung. Es gibt vier Kombinationsmöglichkeiten:

7	6	5	4	3	2	1	0
TODIN 50Hz 60 Hz	externer Signalverkehr	in MODE	Force-load	ONE Shot / Continuous	externer Signalverkehr		Start Stop

Bild 49. Das Kontrollregister des Timer A.

7	6	5	4	3	2	1	0
ALARM	In MODE		Force-load	ONE Shot / Continuous	externer Signalverkehr		Start Stop

Bild 50. Dasselbe für den Timer B.

Register Name	Nr.	6	7	5	4	3	2	1	0
TOD10TH	08			unbenutzt		⅒-Sekundenwert			
TODSEC	09		unbenutzt	Zehnerstelle Sekunden		Einerstelle Sekunden			
TODMIN	0A		unbenutzt	Zehnerstelle Minunten		Einerstelle Minuten			
TODHR	0B		AM/PM Flagge	unbenutzt	Zehnerstelle Stunden	Einerstelle Stunden			

Bild 51. Die Register der Echtzeituhren.

Bit 6 - Bit 5 Der Timer B wird - wie vorhin der Timer A - im

0 - 0 Systemtakt heruntergezählt.

0 - 1 Der Timer B wird durch externe Signale heruntergezählt.

1 - 0 Der Timer B zählt die Unterläufe von Timer A. Das ist der vorhin erwähnte Punkt, der beide Timer kombiniert zum 32-Bit-Zähler. Man kann also im Extremfall 65536 mal 65536 Takte zählen lassen.

1 - 1 Auch in diesem Fall zählt Timer B die Unterläufe von Timer A. Er tut das aber nur, wenn ein bestimmtes externes Signal vorhanden ist.

Bit 7: Auch beim Register CRB steuert dieses Bit bestimmte Möglichkeiten der Echtzeituhr. Deshalb haben Sie noch ein wenig Geduld, bis wir diese Uhr behandeln.

Wir kennen uns nun ganz gut aus, wie wir mit den Timern umzugehen haben. Unser Wissen soll in einem kleinen Test erprobt werden. Dazu bedienen wir uns des $\frac{1}{60}$-Sekunden-IRQ. Wir verändern diese regelmäßige Unterbrechung derart, daß sie nur noch einmal in der Sekunde geschieht. Welche Zahlen dazu in ein 32-Bit-Register gepackt werden müssen, haben wir schon vorhin berechnet. Jeweils in der Reihenfolge LSB/MSB müssen wir sie einschreiben und vorher die Timer anhalten, indem die Bits 0 der Kontrollregister CRA und CRB auf 0 gesetzt werden. Nach dem Einschreiben und Starten der beiden Timer müssen folgende Bitmuster in CRA und CRB stehen:

CRA
- Bit 0 = 1 Start Timer A
- Bit 3 = 0 Dauerlauf
- Bit 5 = 0 Systemtakt

CRB
- Bit 0 = 1 Start Timer B
- Bit 3 = 0 Dauerlauf
- Bit 5 = 0
- Bit 6 = 1 Timer B zählt Unterläufe von Timer A.

Bevor wir die Timer starten, muß auch noch das Interrupt-Kontrollregister verändert werden (das hatten wir uns in Kap. 52 genauer angesehen). Bislang erzeugt ein Unterlauf des Timer A eine Unterbrechung. Wir möchten aber, daß der Timer B (damit wir das 32-Bit-Register voll ausnutzen) der Auslöser ist. Dazu muß Bit 0 des ICR gelöscht und statt dessen Bit 1 gesetzt werden.

```
programm : prg.timer-testc000 c051
----------------------------------

c000 : 78 ad 0e dc 29 fe 8d 0e  4b
c008 : dc ad 0f dc 29 fe 8d 0f  f9
c010 : dc a9 0f 8d 06 dc a9 00  24
c018 : 8d 07 dc a9 a0 8d 04 dc  d5
c020 : a9 08 8d 05 dc a9 1f 8d  84
c028 : 0d dc a9 82 8d 0d dc ad  6e
c030 : 0e dc 29 d7 8d 0e dc ad  0a
c038 : 0f dc 29 d7 8d 0f dc ad  1b
c040 : 0e dc 09 01 8d 0e dc ad  37
c048 : 0f dc 09 41 8d 0f dc 58  a5
c050 : 60 ff 00 ff 00 ff 00 ff  b0
```

Listing 8. Programm Timer-Test, ein Beispiel für die Anwendung eines 32-Bit-Timers.

Im Programm »Timer-Test« (siehe Listing 8 und 9) ist all das realisiert. Mit SYS 49152 gestartet, zeigt sich sofort ein deutlich verlangsamter Cursor. Noch langsamer kann alles werden, indem Sie höhere Werte in die Timer-Register schreiben. Den Normalzustand stellen Sie einfach durch Drücken der RUN/STOP- und der RESTORE-Tasten her. Dabei wird ja - wie Sie aus den letzten Kapiteln her wissen, auch ein I/O-Reset ausgeführt, der den Ausgangszustand wiederherstellt.

Die Verlängerung des IRQ-Zyklus hat übrigens noch einen sinnvollen Nebeneffekt: Je seltener ein laufendes Pro-

```
PASS 1


PASS 2
7000              0823 ;
7000              084C ;*************************************
7000              0375 ;*                                   *
7000              089E ;*            TIMER-TEST             *
7000              08C7 ;*                                   *
7000              08F0 ;* TIMER A UND B DES CIA1 WERDEN SO  *
7000              0919 ;* GESCHALTET, DASS NUR NOCH 1 MAL   *
7000              0942 ;* PRO SEKUNDE DER TIMER-IRQ AUFTRITT *
7000              096B ;*                                   *
7000              0994 ;*    HEIMO PONNATH  HAMBURG  1985   *
7000              09BD ;*                                   *
7000              09E6 ;*************************************
7000              09E9 ;
7000              09EC ;
C000              09F9              .BA $C000
C000              09FE              .OS
C000              0A01 ;
C000              0A27 ;+++ BENUTZTE ADRESSEN DES CIA 1 +++
C000              0A2A ;
C000              0A3A TALO         .DE $DC04
C000              0A4A TAHI         .DE $DC05
C000              0A5A TBLO         .DE $DC06
C000              0A6A TBHI         .DE $DC07
C000              0A79 ICR          .DE $DC0D
C000              0A88 CRA          .DE $DC0E
C000              0A97 CRB          .DE $DC0F
C000              0A9A ;
C000              0AC3 ;+++ EINSCHALTEN DES 1 SEKUNDEN IRQ +++
C000              0AC6 ;
C000 78           0AE5 START        SEI               ;SPERREN ALLER IRQS
C001              0AE8 ;
C001 AD 0E DC     0AF2              LDA CRA
C004 29 FE        0B03              AND #%11111110
C006 8D 0E DC     0B1B              STA CRA           ;STOP TIMER A
C009 AD 0F DC     0B25              LDA CRB
C00C 29 FE        0B36              AND #%11111110
C00E 8D 0F DC     0B4E              STA CRB           ;STOP TIMER B
C011              0B51 ;
C011 A9 0F        0B6F              LDA #15           ;NEUER STARTWERT IN
C013 8D 06 DC     0B8A              STA TBLO          ;32-BIT-REGISTER
C016 A9 00        0B94              LDA #00
C018 8D 07 DC     0B9F              STA TBHI
C01B A9 A0        0BAA              LDA #160
C01D 8D 04 DC     0BB5              STA TALO
C020 A9 08        0BBF              LDA #08
C022 8D 05 DC     0BCA              STA TAHI
C025              0BCD ;
C025 A9 1F        0BDE              LDA #%00011111
C027 8D 0D DC     0BFB              STA ICR           ;ALLE IRQ VERBOTEN
C02A A9 82        0C0C              LDA #%10000010
C02C 8D 0D DC     0C27              STA ICR           ;NUR TIMER B IRQ
C02F              0C2A ;
C02F AD 0E DC     0C34              LDA CRA
C032 29 D7        0C45              AND #%11010111
C034 8D 0E DC     0C61              STA CRA           ;BITS 3 UND 5 = 0
C037              0C64 ;
C037 AD 0F DC     0C6E              LDA CRB
C03A 29 D7        0C7F              AND #%11010111
C03C 8D 0F DC     0C8F              STA CRB           ;DITO
C03F              0C92 ;
C03F AD 0E DC     0C9C              LDA CRA
C042 09 01        0CAD              ORA #%00000001
C044 8D 0E DC     0CC6              STA CRA           ;TIMER A START
C047              0CC9 ;
C047 AD 0F DC     0CD3              LDA CRB
C04A 09 41        0CE4              ORA #%01000001
C04C 8D 0F DC     0D01              STA CRB           ;TIMER B START MIT
C04F              0D24 ;                TIMER A UNTERLAUF
C04F              0D27 ;
C04F 58           0D3D              CLI               ;IRQS FREIGEBEN
C050              0D40 ;
C050 60           0D46              RTS
C051              0D49 ;
C051              0D4F              .EN
```

Listing 9. Der Quelltext zum Timer-Set.

gramm unterbrochen wird, desto schneller wird es mit seinen Jobs fertig. Das kann man immer dann tun – im Extremfall sogar den IRQ ganz ausschalten – wenn man die Möglichkeiten, die der Computer während des normalen IRQ anbietet, nur selten oder aber gar nicht braucht.

65. Die Echtzeituhren

Wir kennen nun fünf Uhren in unserem Computer: Die vier Timer (jeweils A und B im CIA1 und CIA2), die wir, weil wir die Impulszahlen in Zeiteinheiten umrechnen können, zur Zeitmessung einsetzen könnten und die im Basic verfügbare Uhr TI$, die aber – wie wir nun wissen – lediglich die Umsetzung des Timer A im CIA1 in ein bequemer handhabbares Software-Instrument ist. Zudem ist die Ganggenauigkeit dieser Uhr recht gering. Schon einige Kassettenoperationen genügen, sie völlig aus dem Takt zu bringen.

Um so mehr verwundert es, daß zwei hervorragende Echtzeituhren im Commodore 64 so gut wie nie benutzt werden, ja nicht einmal in irgendeiner Weise softwaremäßig unterstützt werden. Vielleicht ist das ein bißchen zuviel »mehr sein als scheinen«, was Commodore da betreibt, wenn man bedenkt, welche verborgenen Schätze da alle zutage gefördert werden können (man denke nur an die hochauflösende Grafik) bei genauer Untersuchung des Computers.

Jeder der beiden CIAs verfügt über solch eine Uhr, die direkt von der Netzfrequenz getaktet wird. Die Zählung der Zeit geschieht in vier Registern (Register $08 bis $0B), die in Bild 51 gezeigt sind.

Vielleicht fällt Ihnen etwas auf, wenn Sie sich diese vier Bytes mal genauer ansehen: Die Speicherung geschieht in Form von Einer- und Zehnerstellen. Das kann also weder im Binärformat noch als ASCII-Zeichen funktionieren. Hier werden die Ziffern als BCD-Zahlen abgelegt. In Kapitel 13 wurde dieses »binary coded decimal«-Format erklärt. Das ist lange her und soll deshalb hier noch mal vorgestellt werden, damit alle wissen, wovon die Rede ist.

In dieser Zahlendarstellung wird jede Dezimalstelle einer Zahl gesondert in eine Binärzahl umgewandelt. Dann ergibt sich der folgende Zusammenhang:

Binär	Dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Das war's! Die anderen möglichen Binärkombinationen (also zum Beispiel 1010 etc.) werden nicht benutzt. Die Zahl 25 beispielsweise lautet im BCD-Format:

```
0010   0101
 ↑      ↑
 2      5
```

Jetzt ist es Ihnen sicherlich verständlich, warum für die Sekunden- und Minuten-Zehnerstellen nicht mehr als drei Bits reserviert wurden: größer als 6 wird die Zehnerstelle nicht.

Zum Stundenregister TODHR ist aber noch etwas zu sagen: Dort ist nur ein Bit reserviert für die Stunden-Zehnerstelle. Die Uhr läuft nicht bis 24 Uhr, sondern lediglich bis 12 Uhr. Zur Unterscheidung, ob vor- oder nachmittags gemeint ist, dient das Bit 7. Dieses sogenannte AM/PM-Flag ist orientiert an der angelsächsischen Gewohnheit, zum Beispiel für 16 Uhr den Ausdruck 4 PM zu verwenden. PM kommt vom lateinischen »post meridiem«, was übersetzt heißt »nach dem Mittag«, wohingegen AM steht für »ante meridiem«, also »vor dem Mittag«. Meint man nun AM, dann muß diese Flagge auf 0, bei PM aber auf 1 gesetzt sein.

Beim Stellen der Uhren sollte eine Reihenfolge eingehalten werden. Sobald nämlich in das Stundenregister geschrieben wird, hält die Zählung automatisch an. Man kann nun die anderen Werte in die Register schreiben. Den Startschuß liefert das Schreiben in das Register TOD10TH: von nun an tickt die Uhr wieder.

Ähnlich funktioniert das Lesen der Uhrzeit. Sobald das Stundenregister gelesen wird, führt das zum Anhalten der Uhr, so daß die restlichen Register reibungslos auslesbar sind. Wieder ist es das Zehntelsekundenregister, das beim Auslesen ein Weiterlaufen der Uhr bewirkt. Aber, so werden Sie bemerken, wenn der Auslesevorgang eine bestimmte Zeit beansprucht, führt das zu Verzögerungen? Die Lösung

ist, daß der gesamte Inhalt der vier Register gleichzeitig mit dem Auslesen des Stundenwertes in einen internen Speicher transferiert wird und dort weiterläuft. Nach dem Lesen des TOD10TH kommt der aktuelle Wert zurück in die Register und dieser wird weitergezählt.

Nun wird es höchste Zeit, daß wir uns die beiden Bits im CRA und im CRB ansehen, die wir vorhin bei der Timer-Behandlung links liegenließen. Bit 7 im CRA kündigt der Echtzeituhr an, welche Netzfrequenz zu erwarten ist. Eine 1 an dieser Stelle steht für 50 Hz, eine 0 für 60 Hz. Unser Stromnetz in Deutschland liefert einen Wechselstrom mit 50 Hz, weshalb wir dann dort die 1 setzen sollten. Da gibt es ein kleines Problem: Beim I/O-Reset, der durch Drücken der RUN/STOP- und der RESTORE-Tasten zusammen ausgelöst wird, schreibt der Computer immer den amerikanischen Wert für 60 Hz in dieses Bit. Dann geht die Uhr aber empfindlich nach. Man muß also einen Weg finden, der erlaubt, dort in diesem Fall wieder eine 1 einzuschreiben. Das ist durch eine eigene NMI-Routine möglich. Sie sehen schon, der Weg zur Nutzung dieser verlockenden Uhren ist dornenreich!

Noch interessanter ist das Bit 7 im CRB. Das Setzen der Uhrzeit ist nämlich nur möglich, wenn dieses Bit den Inhalt 0 hat. Was geschieht, wenn dort eine 1 steht? Dann bestimmt man nicht die aktuelle Uhrzeit, sondern man stellt einen Wecker (das ist die Alarmzeit). Das geschieht nach dem Setzen dieses Bits genauso wie vorhin das Einschreiben der Uhrzeit (also erstaunlicherweise auch in genau dieselben Register!). Im Unterschied dazu ist allerdings ein Lesen der Alarmzeit nicht möglich - das ergibt immer die aktuelle Uhrzeit. Man muß für diesen Fall die Weckzeit irgendwo abspeichern und bei Bedarf dann von dort lesen.

Weil man ja meistens nach dem Erreichen der Alarmzeit irgendeine Reaktion erwartet, ist im ICR (also dem Unterbrechungskontrollregister 13) jedes CIA noch ein Bit reserviert - das Bit 2 —, mit dessen Hilfe der Alarm per IRQ oder NMI wie auch immer geartet losbrechen kann. Der Phantasie sind hier nur wenige Grenzen gesetzt. Wie man mit diesem ICR umgeht, ist Ihnen noch aus der Folge 10 geläufig.

Damit sind wir durch die Eigenheiten der CIAs hindurch. Man braucht tatsächlich keine Scheu zu haben, diese Echtzeituhren zu nutzen. Lediglich die Uhr im CIA1 wird manchmal verwendet, einen bestimmten Wert für die Zufallszahlenerzeugung zu generieren. Aber das sollte einer eigenen Uhren-Routine nicht in die Quere kommen. Solch eine Echtzeituhr finden Sie in Listing 10 und 11.

Befehl	Kapitel
ADC	17
AND	40
ASL	40
BCC	20
BCS	20
BEQ	20
BIT	27
BMI	20
BNE	15,20
BPL	20
BRK	8,49
BVC	20
BVS	20
CLC	17
CLD	13
CLI	49
CLV	27
CMP	23
CPX	23
CPY	23
DEC	12
DEX	12
DEY	12
EOR	40
INC	12
INX	12
INY	12
JMP	28
JSR	28
LDA	8
LDX	10
LDY	10
LSR	41
NOP	27
ORA	40
PHA	34
PHP	34
PLA	34
PLP	34
ROL	41
ROR	41
RTI	49
RTS	9,28
SBC	18
SEC	18
SED	13
SEI	49
STA	9
STX	10
STY	10
TAX	27
TAY	27
TSX	34
TXA	27
TXS	34
TYA	27

Tabelle 22. Alle 6502-Befehle und wo sie besprochen werden

Adressierung	Kapitel
Absolut	9,10
Absolut-X-indiziert	26
Absolut-Y-indiziert	26
Akkumulator	36,40
Implizit	9
Indirekt	28,36
Indirekt-X-indiziert	36
Indirekt-Y-indiziert	36
Relativ	21
Unmittelbar	8,10
Zeropage-absolut	22
Zeropage-absolut-X-indiziert	26
Zeropage-absolut-Y-indiziert	26

Tabelle 23. Alle Adressierungsarten und wo Sie sie finden

Durch SYS49152 aktivieren Sie die Uhr, die Sie mit SYS49261 auch wieder abschalten können. Durch ein USR-Kommando A=USR (String) stellen Sie die Startzeit ein. String kann dabei eine Stringvariable sein oder auch direkt ein String der Form »HHMMSST« (also Stunden, Minuten, Sekunden, Zehntelsekunden). In A steht eine 0, wenn kein Fehler, aber eine -1, wenn ein Fehler aufgetreten ist. Das Lesen der Uhr erfolgt über ein zweites USR-Kommando: PRINTUSR(Zahl). Dabei kann Zahl eine beliebige Zahl oder Variable sein. Eine Alarmzeit ist ebenfalls einstellbar durch ein USR-Kommando, in dem vor der Zeiteingabe noch ein Buchstabe steht. Beispielsweise stellt A=USR(»A1200000«) einen Wecker auf 12 Uhr. Der Alarm im Programm läßt den Bildschirmrahmen blinken. Abstellen kann man das durch Auslösen eines RESTORE-NMI (also RUN/STOP und RESTORE). Sollten Sie vor dem eingestellten Alarm mal solch einen NMI auslösen, dann muß die Alarmzeit neu gestellt werden. Als Basis für dieses Programm diente ein Listing aus dem schon oft erwähnten Buch von Babel/Krause/Dripke »Das Interface Age Systemhandbuch zum Commodore 64«.

Die Unterbrechungs-Programmierung ist damit abgeschlossen – ebenso dieser Kurs, der als Einführung in die Assembler-Alchimie nun alle Geheimnisse der Kunst aufgedeckt hat. In den letzten Kapiteln sind wir schon in die Meistergrade der Zunft aufgestiegen. Vielleicht ging es manchem etwas zu schnell? Dann wird Ihnen der Kurs »Von Basic zu Assembler« eine Hilfe sein, der behutsam und mit vielen an Basic angelehnten Beispielen die nötige Programmierpraxis (ab 64'er, Ausgabe 1/86 sowie im Sonderheft 21). Die Tabellen 22, 23 und 24 sollen Ihnen schnelle Übersichten über Fundstellen und alle Befehle geben. So wie die Segler sich oft »Mast- und Schotbruch« wünschen, verabschiede ich mich, indem ich Ihnen viele grandiose Abstürze wünsche. (Heimo Ponnath/rs)

Befehl	Adressierung	Schema	Bytes	Code Hex	Code Dez	Taktzyklen	Flaggen
ADC	unmittelbar	ADC #Arg	2	69	105	2	N Z C V
	Zeropage-abs.	ADC Arg	2	65	101	3	N Z C V
	Zp-abs-X-ind	ADC Arg,X	2	75	117	4	N Z C V
	absolut	ADC Arg	3	6D	109	4	N Z C V
	abs-X-indiz.	ADC Arg,X	3	7D	125	4	N Z C V
	abs-Y-indiz.	ADC Arg,Y	3	79	121	4	N Z C V
	indir-X-indiz.	ADC (Arg,X)	2	61	97	6	N Z C V
	indir-Y-indiz.	ADC (Arg),Y	2	71	113	5	N Z C V
AND	unmittelbar	AND #Arg	2	29	41	2	N Z
	Zeropage-abs.	AND Arg	2	25	37	3	N Z
	Zp-abs-X-ind	AND Arg,X	2	35	53	4	N Z
	absolut	AND Arg	3	2D	45	4	N Z
	abs-X-indiz.	AND Arg,X	3	3D	61	4	N Z
	abs-Y-indiz.	AND Arg,Y	3	39	57	4	N Z
	indir-X-indiz.	AND (Arg,X)	2	21	33	6	N Z
	indir-Y-indiz.	AND (Arg),Y	2	31	49	5	N Z
ASL	(Akkumulator)	ASL	1	0A	10	2	N Z C
	Zeropage-abs.	ASL Arg	2	06	6	5	N Z C
	Zp-abs-X-ind	ASL Arg,X	2	16	22	6	N Z C
	absolut	ASL Arg	3	0E	14	6	N Z C
	abs-X-indiz.	ASL Arg,X	3	1E	30	7	N Z C
BCC	relativ	BCC Arg	2	90	144	2	—
BCS	relativ	BCS Arg	2	B0	176	2	—
BEQ	relativ	BEQ Arg	2	F0	240	2	—
BIT	Zeropage-abs.	BIT Arg	2	24	36	3	N Z V
	absolut	BIT Arg	3	2C	44	4	N Z V
BMI	relativ	BMI Arg	2	30	48	2	—
BNE	relativ	BNE Arg	2	D0	208	2	—
BPL	relativ	BPL Arg	2	10	16	2	—
BRK	implizit	BRK	1	00	0	7	I
BVC	relativ	BVC Arg	2	50	80	2	—
BVS	relativ	BVS Arg	2	70	112	2	—
CLC	implizit	CLC	1	18	24	2	C
CLD	implizit	CLD	1	D8	216	2	D
CLI	implizit	CLI	1	58	88	2	I
CLV	implizit	CLV	1	B8	184	2	V
CMP	unmittelbar	CMP #Arg	2	C9	201	2	N Z C
	Zeropage-abs.	CMP Arg	2	C5	197	3	N Z C
	Zp-abs-X-ind	CMP Arg,X	2	D5	213	4	N Z C
	absolut	CMP Arg	3	CD	205	4	N Z C
	abs-X-indiz.	CMP Arg,X	3	DD	221	4	N Z C
	abs-Y-indiz.	CMP Arg,Y	3	D9	217	4	N Z C
	indir-X-indiz.	CMP (Arg,X)	2	C1	193	6	N Z C
	indir-Y-indiz.	CMP (Arg),Y	2	D1	209	5	N Z C
CPX	unmittelbar	CPX #Arg	2	E0	224	2	N Z C
	Zeropage-abs.	CPX Arg	2	E4	228	3	N Z C
	absolut	CPX Arg	3	EC	236	4	N Z C
CPY	unmittelbar	CPY #Arg	2	C0	192	2	N Z C
	Zeropage-abs.	CPY Arg	2	C4	196	3	N Z C
	absolut	CPY Arg	3	CC	204	4	N Z C
DEC	Zeropage-abs.	DEC Arg	2	C6	198	5	N Z
	Zp-abs-X-ind	DEC Arg,X	2	D6	214	6	N Z
	absolut	DEC Arg	3	CE	206	6	N Z
	abs-X-indiz.	DEC Arg,X	3	DE	222	7	N Z
DEX	implizit	DEX	1	CA	202	2	N Z
DEY	implizit	DEY	1	88	136	2	N Z
EOR	unmittelbar	EOR #Arg	2	49	73	2	N Z
	Zeropage-abs.	EOR Arg	2	45	69	3	N Z
	Zp-abs-X-ind	EOR Arg,X	2	55	85	4	N Z
	absolut	EOR Arg	3	4D	77	4	N Z
	abs-X-indiz.	EOR Arg,X	3	5D	93	4	N Z
	abs-Y-indiz.	EOR Arg,Y	3	59	89	4	N Z
	indir-X-indiz.	EOR (Arg,X)	2	41	65	6	N Z
	indir-Y-indiz.	EOR (Arg),Y	2	51	81	5	N Z
INC	Zeropage-abs.	INC Arg	2	E6	230	5	N Z
	Zp-abs-X-ind	INC Arg,X	2	F6	246	6	N Z

Tabelle 24. Alle Befehle im Überblick

Befehl	Adressierung	Schema	Bytes	Code Hex	Code Dez	Taktzyklen	Flaggen
	absolut	INC Arg	3	EE	238	6	N Z
	abs-X-indiz.	INC Arg,X	3	FE	254	7	N Z
INX	implizit	INX	1	E8	232	2	N Z
INY	implizit	INY	1	C8	200	2	N Z
JMP	absolut	JMP Arg	3	4C	76	3	—
	indirekt	JMP (Arg)	3	6C	108	5	—
JSR	absolut	JSR Arg	3	20	32	6	—
LDA	unmittelbar	LDA #Arg	2	A9	169	2	N Z
	Zeropage-abs.	LDA Arg	2	A5	165	3	N Z
	Zp-abs-X-ind	LDA Arg,X	2	B5	181	4	N Z
	absolut	LDA Arg	3	AD	173	4	N Z
	abs-X-indiz.	LDA Arg,X	3	BD	189	4	N Z
	abs-Y-indiz.	LDA Arg,Y	3	B9	185	4	N Z
	indir-X-indiz.	LDA (Arg,X)	2	A1	161	6	N Z
	indir-Y-indiz.	LDA (Arg),Y	2	B1	177	5	N Z
LDX	unmittelbar	LDX #Arg	2	A2	162	2	N Z
	Zeropage-abs.	LDX Arg	2	A6	166	3	N Z
	Zp-abs-Y-ind	LDX Arg,Y	2	B6	182	4	N Z
	absolut	LDX Arg	3	AE	174	4	N Z
	abs-Y-indiz.	LDX Arg,Y	3	BE	190	4	N Z
LDY	unmittelbar	LDY #Arg	2	A0	160	2	N Z
	Zeropage-abs.	LDY Arg	2	A4	164	3	N Z
	Zp-abs-X-ind	LDY Arg,X	2	B4	180	4	N Z
	absolut	LDY Arg	3	AC	172	4	N Z
	abs-X-indiz.	LDY Arg,X	3	BC	188	4	N Z
LSR	(Akkumulator)	LSR	1	4A	74	2	N Z C
	Zeropage-abs.	LSR Arg	2	46	70	5	N Z C
	Zp-abs-X-ind	LSR Arg,X	2	56	86	6	N Z C
	absolut	LSR Arg	3	4E	78	6	N Z C
	abs-X-indiz.	LSR Arg,X	3	5E	94	7	N Z C
NOP	implizit	NOP	1	EA	234	2	—
ORA	unmittelbar	ORA #Arg	2	09	9	2	N Z
	Zeropage-abs.	ORA Arg	2	05	5	3	N Z
	Zp-abs-X-ind	ORA Arg,X	2	15	21	4	N Z
	absolut	ORA Arg	3	0D	13	4	N Z
	abs-X-indiz.	ORA Arg,X	3	1D	29	4	N Z
	abs-Y-indiz.	ORA Arg,Y	3	19	25	4	N Z
	indir-X-indiz.	ORA (Arg,X)	2	01	1	6	N Z
	indir-Y-indiz.	ORA (Arg),Y	2	11	17	5	N Z
PHA	implizit	PHA	1	48	72	3	—
PHP	implizit	PHP	1	08	8	3	—
PLA	implizit	PLA	1	68	104	4	N Z
PLP	implizit	PLP	1	28	40	4	alle
ROL	(Akkumulator)	ROL	1	2A	42	2	N Z C
	Zeropage-abs.	ROL Arg	2	26	38	5	N Z C
	Zp-abs-X-ind	ROL Arg,X	2	36	54	6	N Z C
	absolut	ROL Arg	3	2E	46	6	N Z C
	abs-X-indiz.	ROL Arg,X	3	3E	62	7	N Z C
ROR	(Akkumulator)	ROR	1	6A	106	2	N Z C
	Zeropage-abs.	ROR Arg	2	66	102	5	N Z C
	Zp-abs-X-ind	ROR Arg,X	2	76	118	6	N Z C
	absolut	ROR Arg	3	6E	110	6	N Z C
	abs-X-indiz.	ROR Arg,X	3	7E	126	7	N Z C
RTI	implizit	RTI	1	40	64	6	alle
RTS	implizit	RTS	1	60	96	6	—
SBC	unmittelbar	SBC #Arg	2	E9	233	2	N Z C V
	Zeropage-abs.	SBC Arg	2	E5	229	3	N Z C V
	Zp-abs-X-ind	SBC Arg,X	2	F5	245	4	N Z C V
	absolut	SBC Arg	3	ED	237	4	N Z C V
	abs-X-indiz.	SBC Arg,X	3	FD	253	4	N Z C V
	abs-Y-indiz.	SBC Arg,Y	3	F9	249	4	N Z C V
	indir-X-indiz.	SBC (Arg,X)	2	E1	225	6	N Z C V
	indir-Y-indiz.	SBC (Arg),Y	2	F1	241	5	N Z C V
SEC	implizit	SEC	1	38	56	2	C
SED	implizit	SED	1	F8	248	2	D
SEI	implizit	SEI	1	78	120	2	I
STA	Zeropage-abs.	STA Arg	2	85	133	3	—
	Zp-abs-X-ind	STA Arg,X	2	95	149	4	—
	absolut	STA Arg	3	8D	141	4	—
	abs-X-indiz.	STA Arg,X	3	9D	157	5	—
	abs-Y-indiz.	STA Arg,Y	3	99	153	5	—
	indir-X-indiz.	STA (Arg,X)	2	81	129	6	—
	indir-Y-indiz.	STA (Arg),Y	2	91	145	6	—
STX	Zeropage-abs.	STX Arg	2	86	134	3	—
	Zp-abs-Y-ind	STX Arg,Y	2	96	150	4	—

Tabelle 24. Fortsetzung

Befehl	Adressierung	Schema	Bytes	Code Hex	Code Dez	Taktzyklen	Flaggen
	absolut	STX Arg	3	8E	142	4	—
STY	Zeropage-abs.	STY Arg	2	84	132	3	—
	Zp-abs-X-ind	STY Arg,X	2	94	148	4	—
	absolut	STY Arg	3	8C	140	4	—
TAX	implizit	TAX	1	AA	170	2	N Z
TAY	implizit	TAY	1	A8	168	2	N Z
TSX	implizit	TSX	1	BA	186	2	N Z
TXA	implizit	TXA	1	8A	138	2	N Z
TXS	implizit	TXS	1	9A	154	2	—
TYA	implizit	TYA	1	98	152	2	N Z

Erläuterungen: Befehle ADC, AND, EOR, LDA, LDX, LDY und SBC benötigen jeweils einen Taktzyklus mehr, wenn eine Pagegrenze überschritten wird. Für alle Branch-Befehle gilt bei Eintreten der Sprungbedingung: Zu den angegebenen Taktzyklen sind 2 hinzuzuzählen, wenn innerhalb einer Page gesprungen wird und 3, wenn der Sprung in eine andere Page erfolgt.

Tabelle 24, Fortsetzung

```
Name : alarmuhr.ob         c000 c1a1
-----------------------------------
c000 : a9 8e 8d 11 03 a9 c0 8d  11
c008 : 12 03 a9 1d 8d 18 03 a9  a3
c010 : c0 8d 19 03 ad 0e dd 09  12
c018 : 80 8d 0e dd 60 48 8a 48  a1
c020 : 98 48 a9 7f 8d 0d dd ac  49
c028 : 0d dd 10 06 4c 7e c1 4c  41
c030 : 72 fe 20 bc f6 20 e1 ff  b9
c038 : d0 f5 a2 04 bd 2f fd 9d  b4
c040 : 13 03 ca d0 f7 a2 1a bd  1a
c048 : 35 fd 9d 19 03 ca d0 f7  c0
c050 : a9 7f 8d 0d dc 8d 0d dd  e8
c058 : 8d 00 dc a9 08 8d 0e dc  30
c060 : a9 88 8d 0e dd a9 08 20  fe
c068 : b6 fd 4c 6c fe a9 48 8d  37
c070 : 11 03 a9 b2 8d 12 03 78  2a
c078 : a9 47 8d 18 03 a9 fe 8d  c0
c080 : 19 03 a9 31 8d 14 03 a9  84
c088 : ea 8d 15 03 58 60 24 0d  12
c090 : 30 03 4c 34 c1 20 82 b7  72
c098 : c0 07 d0 42 ad 0f dd 29  75
c0a0 : 7f 8d 0f dd a0 00 a9 24  5e
c0a8 : 20 12 c1 c9 00 d0 02 a9  5d
c0b0 : 24 c9 13 90 07 f8 38 e9  7d
c0b8 : 12 d8 09 80 8d 0b dd 20  72
c0c0 : 10 c1 8d 0a dd 20 10 c1  f8
c0c8 : 8d 09 dd 20 7a c1 8d 08  51
c0d0 : dd a9 00 4c 3c bc 68 68  28
c0d8 : 68 68 a9 ff d0 f5 c0 08  ae
c0e0 : d0 f8 ad 0f dd 09 80 8d  bd
c0e8 : 0f dd a9 84 8d 0d dd a9  ed
c0f0 : 3c 85 04 85 02 a9 ff 85  19
c0f8 : 03 a0 01 a9 24 20 12 c1  d0
c100 : c9 00 d0 04 a9 12 d0 b4  56
c108 : c9 12 f0 ae 90 ae b0 a5  79
c110 : a9 60 85 24 20 27 c1 0a  26
c118 : 0a 0a 0a 85 25 20 27 c1  ce
c120 : 05 25 c5 24 b0 b2 60 b1  33
c128 : 22 38 e9 30 90 a8 c9 0a  70
c130 : b0 a4 c8 60 a9 07 20 7d  bf
c138 : b4 a0 00 ad 0b dd 08 29  04
c140 : 1f c9 12 d0 02 a9 00 28  a0
c148 : 10 05 f8 18 69 12 d8 20  e7
c150 : 69 c1 ad 0a dd 20 69 c1  4e
c158 : ad 09 dd 20 69 c1 ad 08  71
c160 : dd 20 74 c1 68 68 4c ca  33
c168 : b4 48 4a 4a 4a 4a 20 74  7d
c170 : c1 68 29 0f 09 30 91 62  af
c178 : c8 60 20 27 c1 60 a9 8b  3a
c180 : 8d 14 03 a9 c1 8d 15 03  f0
c188 : 4c bc fe c6 02 f0 03 4c  17
c190 : 31 ea a5 04 85 02 ad 20  80
c198 : d0 45 03 8d 20 d0 4c 31  99
c1a0 : ea c0 90 ea 8d 2a c2 20  e1
```

Listing 10. Eine Echtzeituhr.

```
10  -;************************************************
20  -;*                                              *
30  -;* Echtzeituhr mit Alarmfunktion                *
40  -;*                                              *
50  -;* Laeuft mit dem NMI-CIA                       *
60  -;* in Verbindung mit dem IRQ fuer               *
70  -;* den Alarm                                    *
80  -;*                                              *
90  -;*  Heimo Ponnath  Hamburg  1985                *
100 -;*                                              *
110 -;* (Teilweise wurde ein Programm aus            *
120 -;* dem Interface Age Systemhandbuch             *
130 -;* zum Commodore 64, Seite 114                  *
140 -;* als Basis verwendet)                         *
150 -;*                                              *
160 -;************************************************
170 -;*                                              *
180 -;* Korrigiert von:                              *
190 -;*                                              *
200 -;* Uwe Thiem                                    *
210 -;* Kasernenstrasse 10                           *
220 -;* 3300 Braunschweig                            *
230 -;* September 1988                               *
240 -;*                                              *
250 -;************************************************
260 -;
280 -;
290 -                     .base $c000
300 -                     .object "2.alarm.ob"
340 -;*********** Zeropage-Labels ********************
350 -;
360 -.define verz             := $02  ;aktuelle
                                     Verzoegerung
370 -.define farb             := $03  ;Wert fuer
                                     Rahmen-EOR-
                                     Operation
380 -.define vorw             := $04  ;Verzoegerungs-
                                     wert
390 -.define valtyp           := $0d  ;Inhalt:ff =
                                     Str  0=N
400 -.define index            := $22
410 -.define index3           := $24  ;Pointer
420 -.define index4           := $25
430 -.define fac1             := $62  ;1.Mantissen-
                                     byte
450 -;*********** Labels Page 3 **********************
460 -;
470 -.define usraddl          := $0311 ;USR-Pointer
480 -.define usraddh          := $0312
490 -.define frei             := $0313
500 -.define irqvl            := $0314 ;IRQ-Vector
510 -.define irqvh            := $0315
520 -.define nminvl           := $0318 ;NMI-Vector
530 -.define nminvh           := $0319
540 -;
550 -;*********** Labels Interpreter *****************
560 -;
570 -.define illquerr         := $b248 ;illegal
                                      quantity-
                                      error =
                                      Normalwert
                                      USR-Vector
580 -.define strini8          := $b47d ;Speicherplatz
                                      pruefen,
                                      String-
                                      pointer setzen
590 -.define strlit67         := $b4ca ;Rest der
                                      String-
                                      leseroutine
600 -.define len1             := $b782 ;String-
                                      laenge in
                                      Y-Register
```

```
610  -.define actofc          := $bc3c ;Akku nach FAC
620  -;
630  -;*********** Labels VIC-II-Chip *****************
640  -;
650  -.define rand            := $d020 ;Rahmenfarbe
660  -;
670  -;*********** Labels CIA-Chips ******************
680  -;
690  -.define cia1            := $dc00 ;Start CIA 1
700  -.define icr1            := $dc0d ;IRQ-Kontroll-
                                        register
710  -.define cra1            := $dc0e ;Timer A
                                        Kontroll-
                                        register
720  -;
730  -.define tod10th2        := $dd08 ;1/10 Sekunden
740  -.define todsec2         := $dd09 ;Sekunden
750  -.define todmin2         := $dd0a ;Minuten
760  -.define todhr           := $dd0b ;Stunden
                                        + AM/PM-Flag
770  -.define icr2            := $dd0d ;NMI-Kontroll-
                                        register
780  -.define cra2            := $dd0e ;Timer A
                                        Kontroll-
                                        register
790  -.define crb2            := $dd0f ;Timer B
                                        Kontroll-
                                        register
800  -;
810  -;*********** Labels oberes ROM *****************
820  -;
830  -.define norm            := $ea31 ;normaler IRQ
840  -.define tastflag        := $f6bc ;Teil der
                                        NMI-Routine
                                        (kein Modul)
850  -.define vectab          := $fd2f ;Tabelle der
                                        ROM-Vectoren
860  -.define vectab7         := $fd35 ;MSB des NMI-
                                        Vectors in der
                                        Tabelle
870  -.define ioreset19       := $fdb6 ;I/O-Reset: bei
                                        Setzen des CRA
                                        im IRQ-CIA
880  -.define nmixct16        := $fe6c ;NMI-Routine ab
                                        Screen-Editor-
                                        Reset
890  -.define nmirs232        := $fe72 ;NMI-Routine ab
                                        RS232-Handling
900  -.define nmiend          := $febc ;Ende der NMI-
                                        Routine
910  -.define stop            := $ffe1 ;Kernal Stop
                                        Sprung nach
                                        JMP ($0328)
10000-;************ Aktivieren ***********************
10010-;
10020-init          lda #<(usr)   ;USR-Vector
                                   laden
10030-              sta usraddl
10040-              lda #>(usr)
10050-              sta usraddh
10060-;
10070-              lda #<(nmi)   ;NMI-Vector
                                   mit
10080-              sta nminvl    ;Startadresse
10090-              lda #>(nmi)   ;der eigenen
10100-              sta nminvh    ;NMI-Routine
                                   laden
10110-;
10120-              lda cra2      ;Bit 7 CRA
                                   setzen
10130-              ora #%1000'0000
10140-              sta cra2      ;Netzfrequenz =
                                   50 Hz
10160-              rts
10170-;
10180-;
10190-;*********** Eigene NMI-Routine ****************
10200-;
10210-nmi           pha             ;Anfang normale
                                     NMI-Routine
10220-              txa             ;Register ret-
ten
10230-              pha
10240-              tya
10250-              pha
10260-;
10270-              lda #$7f        ;Sperren aller
                                     NMI
10280-              sta icr2
10290-;
10300-              ldy icr2        ;pruefen ob NMI
10310-              bpl restnmi     ;vom CIA 2
                                     kommt,
                                     wenn nein,
                                     dann Sprung
10320-              jmp alarm       ;wenn ja, dann
                                     Alarm
10330-cianmi        jmp nmirs232    ;rest der
                                     normalen NMI-
                                     Routine
10340-;
10350-;*********** Eigene Restore-NMI-Routine *********
10360-;
10370-restnmi       jsr tastflag    ;Teil der NMI-
10380-              jsr stop        ;Routine zur
                                     STOP-
10390-              bne cianmi      ;Tasten-Abfrage
10400-;
10410-              ldx #$04        ;IRQ- und BRK-
                                     Vectoren
10420-umlad1        lda vectab,x    ;restaurieren
10430-              sta frei,x
10440-              dex
10450-              bne umlad1
10460-;
10470-              ldx #$1a        ;restaurieren
                                     der
10480-umlad2        lda vectab7,x   ;restlichen
10490-              sta nminvh,x    ;Vectoren
10500-              dex
10510-              bne umlad2
10520-;
10530-              lda #$7f        ;sperren aller
10540-              sta icr1        ;IRQ
10550-              sta icr2        ;NMI
10560-              sta cia1        ;Datenregister
                                     Port A auf
                                     Normalwert
10570-              lda #$08        ;Timer A
10580-              sta cra1        ;im CIA1
10590-;
10600-              lda #$88        ;Timer A im
                                     CIA2:
10610-              sta cra2        ; Bit 0 auf
                                     Stop
10620-;             Bit 3 auf Einzellauf
10630-;             Bit 5 Systemtakt ein
10640-;             Bit 7 Echtzeituhr = 50
                    Hz
10650-;
10660-              lda #$08
10670-              jsr ioreset19
10680-;
10690-              jmp nmixct16    ;Rest der
                                     normalen
                                     Restore-NMI-
                                     Routine
10730-;******* Abschalten der Time of Day Uhr ***********
10740-;
10750-aus           lda #<(illquerr);USR-Vector
10760-              sta usraddl     ;auf Normalwert
10770-              lda #>(illquerr)
10780-              sta usraddh
```

Listing 11. Der Quelltext zur Echtzeituhr.
Fortsetzung auf der nächsten Seite

```
10790-;
10800-                  sei
10810-                  lda #$47        ;restaurieren
                                         des
10820-                  sta nminvl      ;NMI-Vectors
10830-                  lda #$fe
10840-                  sta nminvh
10850-;
10860-                  lda #<(norm)    ;restaurieren
                                         des
10870-                  sta irqvl       ;IRQ-Vectors
10880-                  lda #>(norm)
10890-                  sta irqvh
10900-;
10910-                  cli
10920-                  rts
10960-;******* durch USR aufrufbare Routine *************
10970-;
10980-usr               bit valtyp      ;welcher
                                         Variablen-Typ
                                         liegt vor
10990-                  bmi string      ;wenn String,
                                         dann ueber-
                                         springen
11000-                  jmp zahlvar     ;sonst Sprung
11010-;
11020-;
11030-;
11040-;******* Stellen der Echtzeituhr *****************
11050-;
11060-string            jsr len1        ;y=Stringlaenge
11070-                  cpy #$07        ;String=7
                                         Zeichen?
11080-                  bne alset       ;nein, dann
                                         Alarm stellen
11090-;
11100-                  lda crb2        ;Timer B im
                                         CIA 2
11110-                  and #$7f        ;Bit 7 loeschen
11120-                  sta crb2        ;normale
                                         Uhrzeit
11130-;
11140-;
11150-;
11160-;******* Auslesen des Zeit-Strings ***************
11170-;
11180-                  ldy #$00        ;Zaehler auf
                                         Null
11190-stellen           lda #$24        ;BCD 24 Std-
                                         Vergleich
11200-                  jsr ascbcd      ;Zeichentest
                                         und Umwandlung
                                         in BCD
11210-                  cmp #0
11220-                  bne std12       ;Stunden
                                         ungleich null,
                                         dann Sprung
11230-                  lda #$24        ;sonst = BCD 24
11240-std12             cmp #$13        ;Stunden
                                         groesser
                                         als 12?
11250-                  bcc stdset      ;nein, dann
                                         Sprung
11260-pm1               sed             ;sonst davon
                                         BCD 12
11270-                  sec             ;subtrahieren
11280-                  sbc #$12        ;und
11290-                  cld
11300-pm                ora #$80        ;Bit 7 setzen
11310-;
11320-stdset            sta todhr       ;BCD-Stunden
                                         und AM/PM-Flag
                                         in TOD-CIA2
11330-;
11340-                  jsr ascbcd1     ;Zeichentest
                                         und Umwandlung
                                         in BCD
11350-                  sta todmin2     ;Ergebnis in
                                         TOD-Minuten-
                                         register
11360-;
11370-                  jsr ascbcd1     ;dasselbe fuer
11380-                  sta todsec2     ;Sekunden
11390-;
11400-                  jsr test        ;pruefen, ob
                                         1/10 Sekunden=
                                         Zahl
11410-                  sta tod10th2    ;und ins TOD-
                                         Register
11420-;
11430-                  lda #0          ;Kennung fuer
                                         OK
11440-akkufac           jmp actofc      ;Akku zur
                                         Uebergabe ins
                                         Basic in FAC
11480-;********** Fehler aufgetreten *******************
11490-;
11500-fehler            pla             ;jsr-Adressen
                                         vom
11510-                  pla             ;Stapel holen
11530-error             pla
11540-                  pla
11550-;
11560-error1            lda #$ff        ;Fehlerkennung
                                         in
11570-                  bne akkufac     ;Akku und FAC
11610-;********** Alarmzeit einlesen *******************
11620-;
11630-alset             cpy #$08        ; 8 Zeichen ?
11640-                  bne error1      ;nein = Fehler
11650-;
11660-                  lda crb2        ;Alarmbit
11670-                  ora #%10000000;setzen
11680-                  sta crb2
11690-;
11700-                  lda #%10000100;Alarm-NMI
11710-                  sta icr2        ;zulassen
11720-;
11730-                  lda #$3c        ;Verzoegerungs-
11740-                  sta vorw        ;Wert vorgeben
11750-                  sta verz
11760-                  lda #$ff        ;EOR-Wert
11770-                  sta farb        ;vorgeben
11780-                  ldy #$01        ;Buchstabe
                                         ueberlesen
11790-;
11800-                  lda #$24        ;BCD 24 Grenze
11810-                  jsr ascbcd      ;Umwandlung in
                                         BCD
11820-                  cmp #0
11830-                  bne al12        ;Stunden
                                         ungleich 0,
                                         dann Sprung
11840-                  lda #$12        ;sonst = 12
11850-                  bne stdset
11860-al12              cmp #$12
11870-                  beq pm          ;gleich 12,
                                         dann AM/PM-
                                         Flag setzen
11880-                  bcc stdset      ;kleiner12,dann
                                         Sprung
11890-                  bcs pm1         ;groesser 12,
                                         dann Sprung
11900-;
11910-;
11920-;
11930-;********* Umwandlung ASCII in BCD ***************
11940-;
11950-ascbcd1           lda #$60        ;BCD 60 als
                                         Grenze fuer
                                         Pruefung
11970-ascbcd            sta index3
11980-                  jsr test1       ;pruefen, ob
                                         Zahl
```

```
11990-             asl             ;unteren Nibble
12000-             asl             ;nach oben
12010-             asl
12020-             asl
12030-             sta index4      ;und zwischen-
                                    speichern
12040-             jsr test1       ;naechste zahl
12050-             ora index4      ;beide Nibble
                                    OR
12060-             cmp index3      ;unter
                                    Grenzwert?
12070-             bcs error       ;nein = Fehler
12080-;
12090-             rts
12130-;********* Pruefung ob ASCII-Zahl ****************
12140-;
12150-test1        lda (index),y   ;Zeichen in
                                    Akku
12160-             sec
12170-             sbc #$30        ;< ASCII 0 ?
12180-             bcc fehler      ;ja = Fehler
12190-;
12200-             cmp #$0a        ;>= ASCII : ?
12210-             bcs fehler      ;ja = Fehler
12220-             iny             ;Schleifen-
                                    zaehler
                                    erhoehen
12230-             rts
12270-;********* Uhr lesen ****************************
12280-;
12290-zahlvar      lda #$07        ;Stringlaenge
12300-             jsr strini8     ;Schafft 7 Byte
                                    Platz fuer
                                    String
12310-             ldy #$00        ;Zaehler auf 0
12320-             lda todhr       ;Stunde
                                    auslesen
12330-             php             ;Status
                                    zwischen-
                                    speichern
12340-;
12350-             and #$1f        ;loeschen des
                                    AM/PM-Flag
12360-             cmp #$12        ;= BCD 12 ?
12370-             bne no12        ;ungleich, dann
                                    Sprung
12380-             lda #$00        ;sonst statt-
                                    dessen 0
12390-;
12400-no12         plp             ;status
                                    zurueckholen
12410-             bpl am          ;Sprung, wenn
                                    AM/PM nicht
                                    gesetzt
12420-;
12430-             sed             ;sonst addieren
12440-             clc             ;von BCD 12
12450-             adc #$12
12460-             cld
12470-;
12480-am           jsr bcdasc      ;Umwandlung in
                                    ASCII
12490-;
12500-             lda todmin2     ;dasselbe fuer
12510-             jsr bcdasc      ;Minuten
12520-;
12530-             lda todsec2     ;dasselbe fuer
12970-alarm        lda #<(alirq)   ;neuer
                                    IRQ-Vector
12980-             sta irqvl
12990-             lda #>(alirq)
13000-             sta irqvh
13010-;
13020-             jmp nmiend      ;Rest der nor-
                                    malen NMI-
                                    Routine
13030-;
13040-;
13050-;
13060-;************ Alarm-IRQ ***************************
13070-;
13080-alirq        dec verz        ;Zeitzaehler
                                    verringern
13090-             beq blink       ;blinken, wenn
                                    schon 0
13100-;
13110-             jmp norm        ;sonst normaler
                                    IRQ
13120-;
13130-blink        lda vorw        ;Zeitzaehler
                                    zuruecksetzen
13140-             sta verz
13150-;
13160-             lda rand        ;Rahmenfarbe
                                    invertieren
13170-             eor farb
13180-             sta rand
13190-;
13200-             jmp norm        ;zum normalen
                                    IRQ
12540-             jsr bcdasc      ;Sekunden
12550-;
12560-             lda tod10th2    ;dasselbe fuer
12570-             jsr bcdasc1     ;1/10 Sekunden
12580-;
12590-             pla             ;USR-String-
                                    Argument
12600-             pla             ;Ruecksprung
                                    vorbereiten
12610-;
12620-             jmp strlit67    ;setzt den
                                    String
                                    fuer Basic
12660-;************ Umwandlung BCD in ASCII *************
12670-;
12680-bcdasc       pha             ;auf Stapel
                                    zwischen-
                                    speichern
12690-;
12700-             lsr             ;oberen Nibbel
                                    nach unten
12710-             lsr
12720-             lsr
12730-             lsr
12740-;
12750-             jsr bcdasc1     ;in ASCII
                                    umwandeln und
                                    speichern
12760-;
12770-             pla             ;zurueckholen
                                    der BCD-Zahl
12780-             and #$0f        ;loeschen des
                                    oberen Nibble
12790-bcdasc1      ora #$30        ;aus Zahl wird
                                    ASCII
12800-;
12810-             sta (fac1),y    ;eintragen in
                                    Stringtabelle
12820-;
12830-             iny             ;Zaehler
                                    erhoehen
12840-             rts
12880-;************ Rest ASCII-BCD *********************
12890-;
12900-test         jsr test1       ;prueft auf
                                    ASCII-Zahl
12910-             rts
12920-;
12930-;
12940-;
12950-;************ NMI-Reaktion auf Alarm *************-
```

Listing 11. (Schluß)

Die Schritte zum DOKTOR DER MASCHINENSPRACHE

Sind Sie den Kinderschuhen der Assembler-Programmierung entwachsen? Dann begleiten Sie uns auf dem Weg zum »Dr. Assembler«. Sie werden alle Prüfungen sicher mit Auszeichnung bestehen! Dieser praxisnahe Kurs enthält viele Tricks und Kniffe. Sie erfahren, was man in Maschinensprache alles machen kann.

Wer das Optimum an Geschwindigkeit aus seinem Computer herausholen will, kommt an Maschinensprache nicht vorbei. Die Grundlagen zur Maschinenprogrammierung wurden bereits im Kurs »Assembler ist keine Alchimie«, den Sie in diesem Sonderheft finden, geschaffen. Das Thema dieses Artikels ist es nun, die Möglichkeiten von Maschinensprache optimal zu nutzen. Sie erfahren, wie man
a) Programme beschleunigt und
b) Speicherplatz sparen kann.

Dazu werden Ihnen eine Vielzahl von Programmiertechniken, Tips und Tricks vermittelt, die Ihnen die Programmierung erleichtern.



1. Beschleunigungen des Betriebssystems (in Assembler)

Der C 64 muß viele Aufgaben gleichzeitig erledigen: Bearbeiten des Hauptprogramms, Ablauf der Systeminterrupts und Senden des Video-Signals (an den Monitor/Fernseher). Alle diese Funktionen erfordern
- viele Zugriffe auf den Datenbus des Prozessors
- und dadurch Ausführungszeit.

Unser Grundproblem ist nun, wie wir den Computer dazu bewegen, diese Aufgaben nicht (oder nur teilweise) auszuführen.

a) Eingriffe in den Systeminterrupt

Eine detaillierte Beschreibung des Systeminterrupts finden Sie im bereits erwähnten Kurs »Assembler ist keine Alchimie«. Hier möchte ich nur zusammenfassen, was im normalen Interrupt des Betriebssystems geschieht: 60mal in der Sekunde wird das Hauptprogramm verlassen und die Routine ab $EA31 angesprungen. Ist diese abgearbeitet, wird wieder ins Hauptprogramm zurückgesprungen. Während dieser Unterbrechung (»interrupt«) tut sich einiges:
- die RUN/STOP-Taste wird überprüft
- die Tastatur und der Datasettenmotor werden abgefragt

- das Cursor-Blinken wird erledigt
- die interne Uhr (TI$) wird gestellt.

Überlegen wir uns, welche Funktionen verzichtbar sind: Die RUN/STOP-Taste bewirkt nur in Basic-Programmen einen Abbruch, in Assembler müßte sie zum Beispiel über »JSR $FFE1« zusätzlich abgefragt werden. Die interne Uhr findet von Maschinensprache aus praktisch keine Verwendung. Kurz und gut, ein Maschinenprogramm kann auf beide Funktionen verzichten. Dies wird durch ein

```
LDA #$34
STA $0314
```

erreicht. Weil der Computer dadurch entlastet wird, läuft das Hauptprogramm etwas schneller ab.

Die Normaleinstellung erhält man mit

```
LDA #$31
STA $0314
```

Beschleunigungsmethode 1:
Trick: Verkürzung der Interrupt-Routine
Nebenwirkungen: Abfrage der STOP-Taste und interne Uhr entfallen

Können Sie zwischenzeitlich auf die ganze Interrupt-Routine verzichten, genügt ein einziger Befehl:

SEI (»set interrupt«)

Er verhindert grundsätzlich das Auftreten von Interrupts. Die Normaleinstellung bewirkt:

CLI (»clear interrupt«)

Beschleunigungsmethode 2:
Trick: Interrupt total abschalten
Nebenwirkungen: Abfrage von Tastatur, STOP-Taste und Datasette, sowie Cursor und interne Uhr entfallen.

Es gibt aber noch eine Möglichkeit, im Zusammenhang mit dem Systeminterrupt: Von der Adresse $DC05, die als Zähler dient, hängt die Anzahl der Interrupts (in der Regel 60 Aufrufe pro Sekunde) in einer bestimmten Zeit ab. Diese Adresse kann durch Schreibzugriff geändert werden. Schreibt man in $DC05 einen niedrigen Wert (im Extremfall 0), so werden sehr viele Interrupts ausgelöst. Dies macht sich in der Geschwindigkeit der Interrupt-Routine bemerkbar. Cursor und Tastaturabfrage werden sehr schnell, die interne Uhr geht vor, und so weiter. Verwendet man eine eigene, eventuell zeitkritische Interrupt-Routine, kann sie auf diese Weise beschleunigt werden.

Dieser Geschwindigkeitszuwachs geht allerdings auf Kosten des Hauptprogramms, das stark verlangsamt wird. Bei wenigen Interrupts (große Zahl in $DC05) wird es beschleunigt. Die entsprechenden Assemblerbefehle lauten:

```
LDA #$FF
STA $DC05
```

um eine starke Beschleunigung zu bewirken.
Die Normaleinstellung wird durch

```
LDA #$3A
STA $DC05
```

erreicht.

Beschleunigungsmethode 3:
Trick: Anzahl der Interruptaufrufe pro Sekunde ändern

Nebenwirkungen: Bei zu wenigen Aufrufen hinken Uhr, Cursor und Tastaturabfrage nach; bei zu vielen werden sie zu schnell.

b) VIC-Register Nummer

Ist Ihnen schon bei Hypra-Load, beim Arbeiten mit der Datasette und einigen Kopierprogrammen aufgefallen, daß manchmal der Bildschirm abgeschaltet wird (ähnlich wie im FAST-Mode des C 128)? Dies kann man mit einem Vorhang vergleichen, der zwischenzeitlich den Bildschirm verdeckt. Der Bildschirm kann zwar nach wie vor (hinter dem Vorhang) geändert werden (PRINT-Anweisungen werden also ausgeführt), aber sichtbar wird die Wirkung erst, wenn der Vorhang entfernt wird.

Verantwortlich für das Ein-/Ausschalten des Bildschirms ist das VIC-Register Nummer 17:

Bit 4 gesetzt: Bildschirm wird angezeigt
Bit 4 gelöscht: Bildschirm wird abgeschaltet und nimmt Rahmenfarbe an.

Da wir die theoretischen Grundlagen haben, brauchen wir nur noch unser Wissen in Befehle umzusetzen:

Bildschirm abschalten:

```
LDA  $D011   ($D011 ist VIC-Register #17)
AND  #$EF    ($EF = %11101111)
STA  $D011              ↑
                      Bit 4
```

In diesem Zustand arbeiten manche Kopierprogramme um zirka 15% schneller. Programme, die nicht auf externe Geräte wie die Floppy zugreifen, laufen zirka 5% schneller ab.

Bildschirm wieder einschalten:

```
LDA  $D011
ORA  4$10    ($10 = %00010000)
STA  $D011              ↑
                      Bit 4
```

Dies ist der Normalzustand.

> **Beschleunigungsmethode 4:**
> Trick: Bildschirm abschalten
> Nebenwirkungen: Der Bildschirminhalt ist nicht zu sehen, geht aber auch nicht verloren.

c) Hinweise zum bisher Gesagten

Alle bis zu dieser Stelle genannten Tricks beziehen sich auf die Beschleunigung von Programmen. Sie lassen sich leicht nachträglich einfügen, weil am Programmalgorithmus keine Änderungen erforderlich sind.

Sie können das Abschalten des Bildschirms mit dem Abschalten oder Einschränken des Interrupts verknüpfen, um die Geschwindigkeit noch weiter zu erhöhen. Wenn Sie den Interrupt ganz abschalten (SEI), bringt es keinen zusätzlichen Gewinn, ihn einzuschränken oder die Zahl der Aufrufe zu ändern.

Beachten Sie bitte, daß alle beschriebenen Tricks durch RUN/STOP-RESTORE, einem Reset oder den Assembler-Befehl BRK rückgängig gemacht werden.

2. Systembeschleunigungen in Basic

Hier erfahren Sie, wie sich die Systembeschleunigungen von Basic aus verwerten lassen. Die Nebenwirkungen bleiben allerdings die gleichen, wie unter 1. genannt.

a) Interrupt einschränken

POKE 788,52 verkürzt die Interrupt-Routine um das Abfragen der RUN/STOP-Taste und das Stellen von TI$.

POKE 788,49 Normalzustand

In Basic ist das Ausfallen von RUN/STOP und TI$ wesentlich störender als in Maschinensprache. Überprüfen Sie daher Ihre Programme auf Verwendung von TI$ und fügen Sie den POKE erst nach (!) der Fertigstellung des Programms ein.

b) Interrupt abschalten

POKE 56334,PEEK (56334) AND 254

schaltet den Interrupt ab,

POKE 56334,PEEK (56334) OR 1

schaltet ihn wieder ein. Dies geschieht dadurch, daß der Timer ab- beziehungsweise wieder eingeschaltet wird.

c) Anzahl der Interrupt-Aufrufe ändern

POKE 56325,0: Extrem viele Interruptaufrufe
POKE 56325,255: Extrem wenige (daraus folgt: Interrupt langsam, Basic-Programm schnell)

d) Bildschirm abschalten

POKE 53265,PEEK(53265) AND 239

schaltet den Bildschirm ab.

```
90 GOTO 200                                   <026>
100 REM >> UP - SCHLEIFE <<                   <138>
110 :                                         <086>
120 PRINT" <TASTE>";:WAIT 198,1:POKE 198,0
    :FOR I=1 TO 7:PRINT CHR$(20);:NEXT        <221>
130 :                                         <106>
140 FOR I=1 TO 100:NEXT                       <122>
150 TI$="000000":FOR I = 0 TO 255:POKE 532
    80,I AND 15:NEXT:PRINT TI$:RETURN         <205>
160 :                                         <136>
170 REM >> UP - CURSORBLINKEN AUS <<          <206>
180 :                                         <156>
190 POKE 207,0:POKE 204,1:PRINT" ":RETURN     <186>
200 REM --------------------                  <222>
210 REM -- HAUPTPROGRAMM --                   <045>
220 REM --------------------                  <242>
230 :                                         <206>
240 PRINT CHR$(147)"DEMO FUER SYSTEMBESCHL
    EUNIGUNGEN (BASIC)";                      <061>
250 PRINT"-----------------------------------
    ---------"                                <127>
260 PRINT"{2DOWN}1) NORMALZUSTAND";:GOSUB
    100                                       <033>
270 :                                         <248>
280 PRINT"{DOWN}2) VERKUERZTER INTERRUPT";
    :POKE 788,52:GOSUB 100:POKE 788,49        <084>
290 :                                         <012>
300 PRINT"{DOWN}3) HAEUFIGE INTERRUPTS";:P
    OKE 56325,20:POKE 204,0:GOSUB 100:GOSU
    B 170                                     <018>
310 :                                         <032>
320 PRINT"4) SELTENE{2SPACE}INTERRUPTS";:P
    OKE 56325,150:POKE 204,0:GOSUB 100:GOS
    UB 170                                    <113>
330 SYS 64931:REM NORMALZUSTAND EIN           <253>
340 :                                         <062>
350 PRINT"5) BILDSCHIRM ABGESCHALTET ";:PO
    KE 53265,PEEK(53265) AND 239:GOSUB 140    <107>
360 POKE 53265,PEEK(53265) OR 16:PRINT"{DO
    WN}** ENDE **"                            <066>
```

© 64'er

Listing 1. Systembeschleunigungen in Basic

```
100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER BASOUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL BASOUT = $FFD2
170  -;
180  -           .LDX #0
190  -SCHLEIFE  LDA TEXT,X      ; ZEICHEN LESEN
200  -           INX
210  -           JSR BASOUT      ; UND AUSGEBEN
220  -           BNE SCHLEIFE    ; SCHON ENDMARKIERUNG?
230  -;
240  -; TEXTAUSGABE (UEBER STROUT)
250  -;
260  -.GL STROUT = $AB1E
270  -;
280  -           LDA #<(TEXT)    ; LOW-BYTE IN AKKU
290  -           LDY #>(TEXT)    ; HIGH-BYTE IN Y
300  -           JMP STROUT      ; TEXTAUSGABE UND ENDE
310  -;
320  -TEXT      .TX "DAS IST DER TEXT!"
330  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 2. Die unkomfortable Lösung, einen Text auszugeben

POKE 53265,PEEK(53265) OR 16
schaltet ihn wieder ein.

An dieser Stelle sei noch einmal auf Punkt 1c hingewiesen, damit keine (vermeidbaren) Probleme auftreten.

Anhand von Listing 1 wollen wir uns nun mit der Anwendung der Systembeschleunigungen befassen. Dieses kleine Beispielprogramm, an dem Sie nach Herzenslust experimentieren können, versucht, mit Hilfe von TI$ die Arbeitsdauer der Schleife (Zeile 150) zu messen.

Während des Ablaufs dieser Schleife, die kontinuierlich die Rahmenfarbe ändert, sollten Sie keine Taste drücken, um die Meßwerte nicht zu verfälschen.
Wenn Sie dies beachten, erhalten Sie folgende Werte:

1. Normalzustand: 000003
2. Verkürzter Interrupt: 000000
   An der gemessenen Zeit können Sie erkennen, daß TI$ abgeschaltet wurde.
3. Häufige Interrupts: 000010
   Aufgrund vieler Interrupt-Anforderungen wurde die Uhr TI$ sehr oft erhöht.
4. Seltene Interrupts: 000001
   Da die IRQ-Routine nur selten durchlaufen wurde, ist TI$ kaum weitergezählt worden.
5. Bildschirm abgeschaltet: 000002

Nur bei diesem Punkt (und natürlich auch bei »1«) hat TI$ volle Aussagekraft bezüglich der Ablaufzeit. An dieser Zeit können wir erkennen, daß durch das Abschalten des Bildschirms tatsächlich gegenüber »1« ein Zeitgewinn anfällt.

Bei den Punkten »3« und »4« wurde der Cursor eingeschaltet. Bei »3« (häufige Interrupts) ist er sehr schnell, bei »4« dagegen sehr langsam.

An Punkt »5« können Sie erkennen, daß bei abgeschaltetem Bildschirm der Hintergrund immer die Rahmenfarbe ($D020) annimmt, ohne daß wir die entsprechende Farbe ins Register $D021 »POKEn«.

3. Optimierung der Bildschirmausgabe

Ohne die Bildschirmausgabe kommt kein Programm aus, aber oft kostet sie unnötig viel Rechenzeit. Der Grund ist hier nicht beim Betriebssystem zu suchen, sondern bei umständlicher Programmierung. Diese wiederum ist auf mangelndes Know-how zurückzuführen, welches wir nun verbessern wollen.

In der Regel wird zur Ausgabe eines Zeichens dieses in den Akku geladen und die Routine BASOUT ($FFD2) aufgerufen. Veranschaulichen wir uns einmal die Arbeitsweise von BASOUT: Das Betriebssystem prüft bei jedem Zeichen, ob es sich um einen Buchstaben oder ein Steuerzeichen, zum Beispiel »Bildschirm löschen« handelt. Buchstaben werden in den Bildschirmcode umgewandelt und ins Bildschirm-RAM ab $0400 geschrieben.

Für Steuerzeichen existieren jeweils Unterroutinen, die zum Beispiel eine Leerzeile einfügen, den Bildschirm löschen oder ähnliches.

Diese aufwendige Überprüfung verlangsamt die Bildschirmausgabe erheblich. BASOUT läßt sich zwar geringfügig beschleunigen, indem man statt bei $FFD2 (Kernel-Einsprung) bei $E716 einsteigt, aber es geht noch schneller:

a) Bildschirm löschen
Langsam:

```
LDA #$93   $93 = 147 = Code für »Bildschirm löschen«, entspricht PRINT CHR$(147)
JSR $FFD2  (oder $E176)
```

Schnell:

```
JSR $E544  (Routine für »Bildschirm löschen«)
```

b) Cursor in Home-Position (linke obere Ecke)
Langsam:

```
LDA #$13 ; $13 = Code für »Cursor Home«
JSR $FFD2  (oder $E176)
```

Schnell:

```
JSR $E566  (Routine für »Cursor Home«)
```

c) Cursor-Positionierung
Langsam:
Senden von Steuerzeichen (CRSR DOWN, UP und so weiter) über BASOUT.
Schnell:

```
LDX #Zeile
LDY #Spalte
JSR $E50C (Cursorposition setzen)
```

d) Textausgabe
Unkomfortable Lösung:
Senden von Zeichen (Buchstaben, Grafikzeichen) über BASOUT.

Eine solche Schleife finden Sie in Listing 2, Zeilen 148 - 220 und 320 - 330. Nach dem Start durch »SYS 49152« gibt Listing 2 zweimal hintereinander den Text »DAS IST DER TEXT« aus. Das erste Mal wird der Text über eine BASOUT-Schleife gedrückt, beim zweiten Mal nimmt das Programm die komfortable Lösung:

```
SEARCHING FOR $$

100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER STROUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL STROUT = $AB1E
170  -.GL CURSOR = $E50C
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -          JSR CLRSCR     ; = PRINT CHR$(147)
240  -          LDX #ZEILE     ; ZEILE IN X
250  -          LDY #SPALTE    ; SPALTE IN Y
260  -          JSR CURSOR     ; CURSOR SETZEN
270  -          LDA #<(TEXT)   ; LOW-BYTE IN AKKU
280  -          LDY #>(TEXT)   ; HIGH-BYTE IN Y
290  -          JMP STROUT     ; TEXTAUSGABE & ENDE
300  -;
310  -TEXT      .TX "DAS IST DER TEXT!"
320  -.BY 0 ; ENDMARKIERUNG FUER STROUT
```

Listing 3. Die komfortable Lösung einen Text auszugeben

Ab der Adresse »TEXT« muß der Text (in ASCII-Darstellung) stehen, in dem keine Anführungszeichen vorkommen dürfen. Am Ende des Textes muß $00 als Endmarkierung zu finden sein. Die Ausgabe erfolgt dann über

```
LDA #<(TEXT)   Low-Byte der Adresse
LDY #>(TEXT)   High-Byte
JSR $AB1E
```

Die Routine $AB1E wird fortan als »STROUT« (STRing-OUTput = String-Ausgabe) bezeichnet. STROUT ist zwar etwas langsamer als BASOUT; dafür erlaubt die komfortable Parameterübergabe eine wesentlich bequemere Programmierung, wie Sie am zweiten Teil von Listing 2 (Zeilen 260 - 300, 320 - 330) sehen können. Mit nur drei Befehlen wird der Text ausgegeben!

Eine Anwendung von (fast) allen Routinen aus der Beschleunigungsmethode 5 zeigt Listing 3.

Der Bildschirm wird gelöscht und in Zeile 12 ab Spalte 10 ein Text ausgegeben. Auch in diesem Programm sollten Sie zur Übung etwas experimentieren!

Beschleunigungsmethode 5.
Zusammenfassung der bisherigen Alternativen zu BASOUT:

CLEAR HOME:	JSR $E544
CURSOR HOME:	JSR $E566
Cursorpositionierung:	LDX #Zeile
	LDY #Spalte
	JSR $E50C
Textausgabe:	Text ab TEXT ablegen
	(wie Listing 2, Zeile 320 - 330)
	LDA #<(TEXT)
	LDY #>(TEXT)
	JSR $AB1E

Alle diese Verfahren sind nicht nur schnell, sondern auch speicherplatzsparend.

e) Kopieren des Textes in den Bildschirmspeicher

Dies ist die schnellste Methode: Der Text wird in den Bildschirmspeicher kopiert. Die lange Umwandlung entfällt völlig, da der Text als fertiger Bildschirmcode im Speicher abgelegt wird. Wenn einige Kopfzeilen (zum Beispiel mit Copyright-Vermerken) an verschiedenen Stellen ausgegeben werden sollen, ist es ratsam, ein kleines Unterprogramm zu erstellen. Dieses schreibt dann die Kopfzeilen direkt in den Bildschirmspeicher, ohne die aktuelle Cursor-Position zu beeinflussen.

```
100  -.LI 1,3,0
110  -;
120  -; TEXT IN VIDEO-RAM SCHREIBEN
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -.GL VIDEORAM = 1024 ; BILDSCHIRMSPEICHER
240  -.GL ADRESSE = VIDEORAM + (40*ZEILE) + SPALTE
250  -;
255  -           JSR CLRSCR     ; = PRINT CHR$(147)
260  -           LDX #0
270  -SCHLEIFE   LDA TEXT,X     ; BILDSCHIRMCODE LESEN
280  -           BEQ ENDE       ; =0, DANN ENDE
290  -           STA ADRESSE,X  ; IN BILDSCHIRMSPEICHER
295  -           INX
296  -           JMP SCHLEIFE   ; NAECHSTES ZEICHEN
300  -ENDE       RTS
305  -;
310  -TEXT       .BY 4,1,19," ",9,19,20," "
311  -.BY 4,5,18," ",20,5,24,20,"!"
320  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 4. Die schnellste Lösung, einen Text auszugeben

Eines müssen Sie aber unbedingt beachten: Die Farbgebung ist nur durch Ändern des Farb-RAMs möglich.

Eine Tabelle der Bildschirmcodes finden Sie übrigens im Anhang des C 64-Handbuchs.

Beschäftigen wir uns nun mit Listing 4:

Dieses Programm entspricht in der Wirkung Listing 3, gibt den Text jedoch nicht über die Betriebssystem-Routinen CURSOR und STROUT aus, sondern schreibt ihn direkt in den Bildschirm.

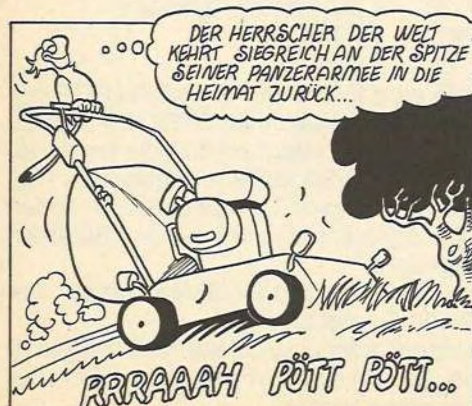
In den Zeilen 310 bis 320 steht der Bildschirmcode des Textes.

Zurück zur Routine STROUT: Diese Routine arbeitet, da sie sich auf die BASOUT-Routine stützt, auch mit Periphe-

```
100  -.LI 1,3,0
110  -;
120  -; DRUCKER-AUSGABE MIT
130  -; DER STROUT-ROUTINE
140  -;
150  -.GL STROUT = $AB1E
160  -.GL SETNAM = $FFBD ; DIE BEDEUTUNG
170  -.GL SETLFS = $FFBA ; DIESER ROUTINEN
180  -.GL OPEN   = $FFC0 ; ENTNEHMEN SIE
190  -.GL CHKOUT = $FFC9 ; BITTE DEM KURS
200  -.GL CLRCHN = $FFCC ; "ASSEMBLER IST
210  -.GL CLOSE  = $FFC3 ; KEINE ALCHIMIE"
220  -;
230  -.MA PRINT (ADRESSE)
240  -           LDA #<(ADRESSE)
250  -           LDY #>(ADRESSE)
260  -           JSR STROUT
270  -.RT
280  -;
290  -.BA $C000 ; START: SYS 49152
300  -;
310  -           LDA #0          ; KEINEN
320  -           JSR SETNAM      ; FILENAMEN
330  -;
340  -           LDA #4          ; LOG. FILENUMMER =4
350  -           TAX             ; GERAETEADRESSE 4
360  -           LDY #0          ; SEKUNDAERADRESSE 0
370  -           JSR SETLFS      ; PARAMETER SETZEN
380  -;
390  -           JSR OPEN        ; FILE OEFFNEN
400  -;
410  -           LDX #4          ; FILENUMMER 4
420  -           JSR CHKOUT      ; AUSGABE AUF DRUCKER LENKEN
430  -;
440  -...PRINT (TEXT) ; TEXT AUSGEBEN
450  -;
460  -           JSR CLRCHN      ; WIEDER BILDSCHIRMAUSGABE
470  -;
480  -...PRINT (TEXT) ; JETZT AUF BILDSCHIRM
490  -;
500  -           LDA #4          ; LOG. FILENUMMER 4
510  -           JMP CLOSE       ; FILE SCHLIESSEN
520  -; & PROGRAMM BEENDEN
530  -;
540  -TEXT       .TX "DIESER TEXT WIRD AUF"
550  -.TX " DEN DRUCKER AUSGEGEBEN !"
560  -.BY 13,13,13,0 ; 3 * CAR.RETURN
```

Listing 5. So gibt man Text auf dem Drucker aus



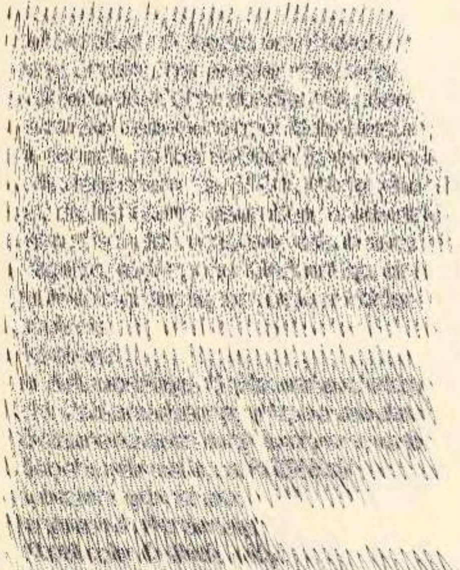


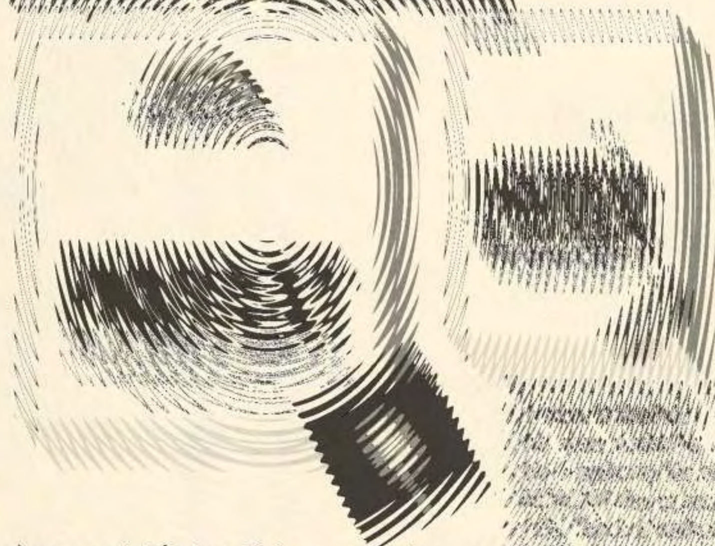




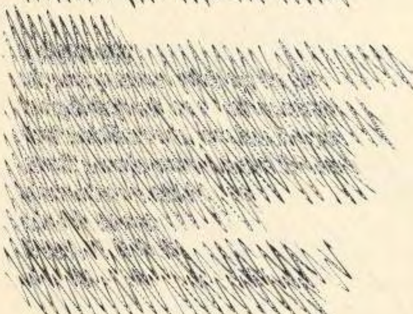


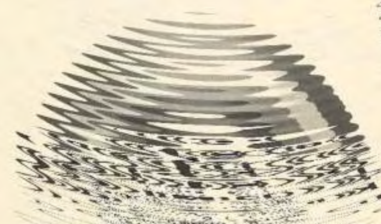

PROTEXT

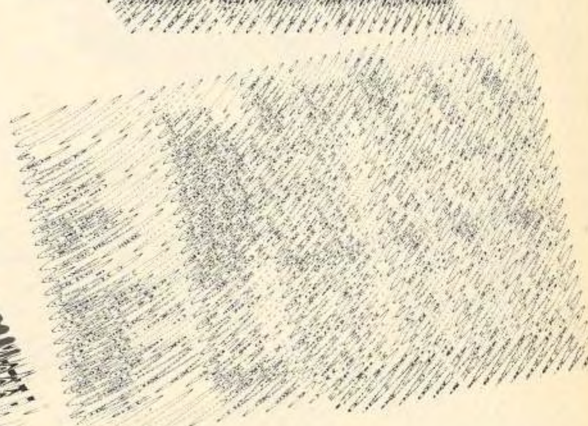












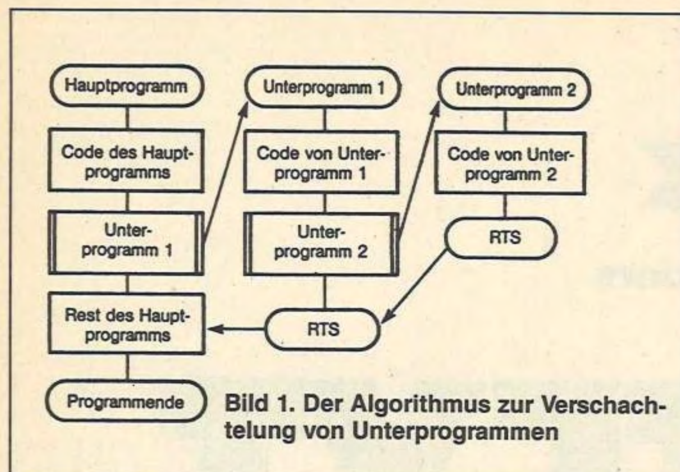


Bild 1. Der Algorithmus zur Verschachtelung von Unterprogrammen

riegeräten wie Floppy und Drucker, wenn diese über dem CMD-Befehl als Ausgabegeräte definiert wurden. In »Assembler ist keine Alchimie« wurde gezeigt, wie man mit der BASOUT-Routine die Drucker-Ausgabe betreibt. Dort wurden alle wichtigen Routinen bis ins Detail beschrieben.

Listing 5 gibt einen Text zuerst auf dem Drucker und dann auf dem Bildschirm aus. Daran soll außer dem Druckerbetrieb auch gezeigt werden, wie man die Parameterübergabe an STROUT als Makro (Zeilen 230 bis 270) definiert und sich somit einen bequemen Ausgabe-Befehl schafft.

4. Unterprogramme

Ohne die Unterprogramm-Befehle JSR und RTS kommt fast kein Maschinenprogramm aus. Es ist allerdings ziemlich unbekannt, daß beide Befehle das Programm stark verlangsamen. Grund genug für uns, JSR und RTS näher zu betrachten:

Trifft der Prozessor auf JSR, schiebt er den aktuellen Programmzähler plus 2 (= Rücksprungadresse - 1) auf den Stack und springt dann zu der Adresse, die hinter JSR steht. Trifft er auf RTS, holt er die Adresse vom Stapel zurück, erhöht sie um 1 und verwendet sie wieder als Programmzähler.

Bemerkenswert ist, daß die Zugriffe auf den Stapel sich in keiner Weise von den Zugriffen über die Befehle PHA und PLA unterscheiden. Daher muß jedesmal der Stapelzeiger neu errechnet werden. Diese vielen Operationen sind schuld daran, daß JSR und RTS so langsam sind.

Da wir das Problem erkannt haben, können wir damit beginnen, unser Wissen anzuwenden.

```
100   -.LI 1,3,0
110   -.BA $C000 ; START: SYS 49152
120   -;
130   -; UNTERPROGRAMMVERSCHACHTELUNG
140   -; (OPTIMIERTE ASSEMBLERVERSION)
150   -;
160   -.GL STROUT = $AB1E
170   -;
180   -.MA PRINT (ADRESSE)
190   -            LDA #<(ADRESSE)
200   -            LDY #>(ADRESSE)
210   -            JSR STROUT
220   -.RT
230   -;
240   -; --------------- HAUPTPROGRAMM
250   -;
260   -...PRINT (TEXT1)
270   -;
280   -            JSR UP1
290   -;           ↑ AUFRUF VON UNTERPROGRAMM 1
300   -;
310   -...PRINT (TEXT2)
320   -;
330   -            JMP $A474      ; WARMSTART
340   -;
350   -;
360   -; --------------- UNTERPROGRAMM 1
370   -;
380   -UP1         NOP            ; BELIEBIGER CODE
390   -...PRINT (TEXT3)
400   -;
410   -;
420   -;
430   -; --------------- CODE VON UNTERPROGRAMM 2
440   -;
450   -UP2         NOP            ; BELIEBIGER CODE
460   -            LDA #<(TEXT4)  ; LOW-BYTE
470   -            LDY #>(TEXT4)  ; HIGH-BYTE
480   -            JMP STROUT     ; TEXTAUSGABE
490   -; UND RUECKSPRUNG VOM UNTERPROGRAMM,
500   -; WEIL AM ENDE DER STROUT-ROUTINE
510   -; EIN RTS-BEFEHL STEHT.
10000-;
10010-;
10020-; --------------- TEXTE
10030-;
10040-TEXT1       .TX "HIER IST DAS HAUPTPROGRAMM."
10050-.BY 13,13 ; 1 LEERZEILE
10060-.BY 0 ; ENDMARKIERUNG
10070-;
10080-TEXT2       .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10090-.BY 13,13,0
10100-;
10110-TEXT3       .TX "HIER IST DAS UNTERPROGRAMM 1."
10120-.BY 13,13,0
10130-;
10140-TEXT4       .TX "HIER IST DAS UNTERPROGRAMM 2."
10150-.BY 13,13,0
```

Listing 6. Die umständliche Methode, Unterroutinen aufzurufen

```
100   -.LI 1,3,0
110   -.BA $C000 ; START: SYS 49152
120   -;
130   -; UNTERPROGRAMMVERSCHACHTELUNG IN ASSEMBLER
140   -;
150   -.GL STROUT = $AB1E
160   -;
170   -.MA PRINT (ADRESSE)
180   -            LDA #<(ADRESSE)
190   -            LDY #>(ADRESSE)
200   -            JSR STROUT
210   -.RT
220   -;
230   -; --------------- HAUPTPROGRAMM
240   -;
250   -...PRINT (TEXT1)
260   -;
270   -            JSR UP1
280   -;           ↑ AUFRUF VON UNTERPROGRAMM 1
290   -;
300   -...PRINT (TEXT2)
310   -;
320   -            JMP $A474      ; WARMSTART
330   -;
340   -;
350   -; --------------- UNTERPROGRAMM 1
360   -;
365   -UP1         NOP            ; BELIEBIGER CODE
370   -...PRINT (TEXT3)
380   -;
390   -            JSR UP2
400   -;           ↑ AUFRUF VON UNTERPROGRAMM 2
410   -;
420   -            RTS            ; UP1 VERLASSEN
430   -;
440   -;
450   -; --------------- UNTERPROGRAMM 2
460   -;
465   -UP2         NOP            ; BELIEBIGER CODE
470   -...PRINT (TEXT4)
480   -;
490   -            RTS            ; UP2 VERLASSEN
500   -;
10000-;
10010-; --------------- TEXTE
10020-;
10030-TEXT1       .TX "HIER IST DAS HAUPTPROGRAMM."
10040-.BY 13,13 ; 1 LEERZEILE
10050-.BY 0 ; ENDMARKIERUNG
10060-;
10070-TEXT2       .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10080-.BY 13,13,0
10090-;
10100-TEXT3       .TX "HIER IST DAS UNTERPROGRAMM 1."
10110-.BY 13,13,0
10120-;
10130-TEXT4       .TX "HIER IST DAS UNTERPROGRAMM 2."
10140-.BY 13,13,0
```

Listing 7. Die optimierte Methode, Unterroutinen aufzurufen

a) Unterprogrammverschachtelung

Stellen wir uns folgendes Beispiel vor: ein Hauptprogramm ruft das Unterprogramm 1 auf. Dieses ruft an seinem Ende das Unterprogramm 2 auf, um dann mit RTS ins Hauptprogramm zurückzukehren.

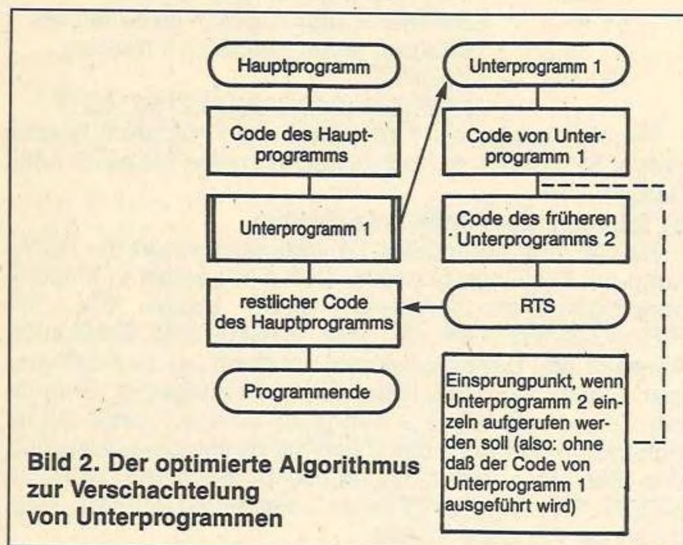


Bild 2. Der optimierte Algorithmus zur Verschachtelung von Unterprogrammen

Alles ziemlich schwierig, oder?

Deshalb gehen wir mit Hilfe einer Grafik vor: In Bild 1 sehen Sie ein Flußdiagramm nach obigem Aufbau. In der Beschriftung soll »Code« nicht »Kennwort« bedeuten, sondern heißt einfach »Befehlsnummer«.

Wie an den Pfeilen zu erkennen ist, werden zwei RTS-Befehle hintereinander abgearbeitet (von Unterprogramm 2 nach Unterprogramm 1 und von dort zum Hauptprogramm). Dies ist immer ein Indiz dafür, daß das Programm noch optimiert werden kann.

Eine »Übersetzung« von Bild 1 in Assembler ist Listing 6: Wenn Sie dieses über »SYS 49152« starten, ist aus den ausgegebenen Texten ersichtlich, welcher Programmteil wann abgearbeitet wird.

Sobald Sie die Struktur von Bild 1 beziehungsweise Listing 6 verstanden haben, können wir uns mit der optimierten Form befassen, die in Bild 2 beziehungsweise Listing 7 zu finden ist.

Hier wird das ehemalige Unterprogramm 2 ans Ende von Unterprogramm 1 gehängt (wobei es ebenfalls über JMP UP2 angesprungen werden könnte). Auf diese Weise muß es nicht über JSR aufgerufen werden, was auch einen RTS-Befehl überflüssig macht.

Trotz dieser Änderung kann das Unterprogramm 2 auch weiterhin als Unterprogramm aufgerufen werden, da bei JSR UP2 die CPU auf einen RTS-Befehl trifft (Bild 2).

In Listing 7 muß noch der JMP-Befehl in Zeile 480 erläutert werden:

Dort muß nicht JSR STROUT:RTS stehen, weil am Ende der STROUT-Routine im ROM ohnehin ein RTS steht. Deshalb benötigt unser Programm keinen eigenen RTS-Befehl zur Rückkehr ins Hauptprogramm.

Die folgende Regel gilt für Aufrufe von Betriebssystemroutinen:

```
JSR  $XXXX      entspricht   JMP $XXXX
RTS
```

Voraussetzung ist, daß im Unterprogramm ab $XXXX keine Stapelmanipulation erfolgt, wie sie gleich beschrieben wird. Das geschilderte Verfahren zur Unterprogrammverschachtelung und die entsprechenden Regeln können Sie dann auf jede (!) Programmiersprache übertragen.

b) Stapelmanipulation

Wenn Sie »Exbasic Level II« kennen, wissen Sie sicher den Befehl »DISPOSE RETURN« zu schätzen. Er dient dazu, ein Unterprogramm ohne RETURN abzuschließen. Dadurch kann dieses zum Beispiel über GOTO verlassen werden.

In Assembler ist dies auch möglich. Die Befehlseingabe

```
PLA
PLA
```

entspricht in der Wirkung »DISPOSE RETURN«.

Da die Rücksprungadresse auf den Stapel abgelegt wird und dort 2 Byte in Anspruch nimmt, kann sie über PLA:PLA wieder vom Stapel geholt werden. Ein Unterprogramm ist nach PLA:PLA eigentlich kein Unterprogramm mehr, sondern Bestandteil des aufrufenden Programms. PLA:PLA findet vor allem in der Fehlerbehandlung Anwendung. An einem späteren Listing werden wir dies noch sehen. Nach PLA:PLA kann ein Unterprogramm über JMP verlassen werden. Dies machen wir uns zunutze, um den Rücksprung an eine beliebige Adresse zu simulieren. Dies ist sonst nicht möglich, da bei RTS immer hinter den Befehl gesprungen wird, der das Unterprogramm aufgerufen hat.

Ein RTS an eine beliebige Adresse müßte »RTS XXXX« heißen, doch diesen Befehl gibt es beim 6510 nicht. So wird er aber simuliert:

```
PLA          ; holt Rücksprungadresse
PLA          ; vom Stapel und
JMP $XXXX    ; springt nach $XXXX
```

So sieht ein Makro dazu aus:

```
-.MA    RTS (RUECKSPRUNGADRESSE)
-       PLA
-       PLA
-       JMP RUECKSPRUNGADRESSE
-.RT
```

Und noch ein Mangel der Unterprogrammbefehle soll beseitigt werden: Obwohl es JMP (indirekt) gibt, kennt der 6510 keinen Befehl wie JSR (indirekt). Über Stapelmanipulation ist dies dennoch möglich (siehe dazu auch im 64'er, Ausgabe 1/86: Assembler-Bedienung leicht gemacht).

Nehmen wir an, im Vektor $14/$15 steht die Adresse $C000. Nun soll über den $14/$15-Vektor ein Unterprogramm aufgerufen werden (also das ab $C000). Bild 3 zeigt, was im einzelnen geschehen muß.

Die Rücksprungadresse steht zwar in Bild 3 direkt hinter dem JMP ($0014)-Befehl, kann aber auch anderswo im Programm stehen.

Folgendes Makro ermöglicht die Simulation von JSR (indirekt):

```
-. MA JSRIND (VEKTOR, RUECKSPRUNGADRESSE)
-          LDA # >(RUECKSPRUNGADRESSE-1)
-          PHA
-          LDA # <(RUECKSPRUNGADRESSE-1)

-          PHA
-          JMP (VEKTOR)
-. RT
```

Diese Simulation von JSR ($XXXX) verwendet auch der SYS-Befehl (disassemblieren Sie von $E12A bis $E155 und betrachten Sie dazu Bild 3).

Zuerst holt er die Zahl nach SYS in die Adressen $14/$15, dann legt er die Rücksprungadresse ($E147) -1 auf dem Stack ab. Nun holt er die Register P, A, X, Y aus den Adressen $030F, $030C, $030D, $030E. Es folgt ein indirekter Sprung über $0014/$0015.

Nach dem Rücksprung werden die Register wieder im

Speicher dort abgelegt, woher sie genommen wurden und ein Sprung ins Basic wird durchgeführt.

Später werden wir noch eine weitere Möglichkeit für JSR (ind) kennenlernen, die aber nicht auf Stapelmanipulation beruht.

c) Vergleich zwischen Unterprogramm und Makro bezüglich Geschwindigkeit

Wenn Sie den Hypra-Ass (Seite 122) oder einen anderen Makro-Assembler besitzen, haben Sie die Möglichkeit, Befehlsfolgen als Makros zu definieren. Makros sind deswegen so beliebt, weil sie den größten Vorteil von Unterprogrammen bieten, nämlich Übersichtlichkeit. Da Makros aber wie »normale« Befehle im Speicher stehen, entfällt der Aufruf über JSR und RTS. Dies ist der Grund, weshalb Makros etwas schneller (wenige Taktzyklen) als Unterprogramme sind. Das Problem, wann Makros und wann Unterprogramme vorteilhaft sind, wird später noch aufgegriffen.

5. Tabellen

Im allgemeinen Sprachgebrauch werden Tabellen als »geordnete Zusammenstellungen von Daten« verstanden. Diese Funktion haben sie auch in Computer-Programmen, wo man sie daran erkennt, daß Tabellen keinen Befehlscharakter haben.

SMON-Benutzer können mit »FT« ein Programm nach Tabellen durchsuchen lassen; dann sucht SMON im Programm nach Bytes, die nicht zu Maschinensprachebefehlen gehören.

Wozu werden nun Tabellen verwendet?

In der Regel dienen Tabellen einem Computerprogramm als »elektronischer Rechenschieber«. So wie das Kopfrechnen durch einen Rechenschieber ersetzt werden kann, weil man nur in einer geordneten Zusammenstellung von Ergebnissen das richtige suchen muß, kann ein Programm aus seinen Tabellen denselben Nutzen ziehen: die Berechnungen entfallen, die Programmierung wird einfacher.

Weniger Berechnungen sind nötig. Daraus entsteht ein deutlicher Geschwindigkeitszuwachs, der Hauptvorteil von Tabellen. Wie man Tabellen einsetzt, erfahren Sie im folgenden.

a) Tabellen aus Rechenergebnissen

Noch einmal zum Rechenschieber. Es geht beim Kopfrechnen viel schneller, 4 x 10 auszurechnen als 4 x 7. Bei einem Rechenschieber besteht kaum ein Unterschied in der »Rechenzeit«.

Dementsprechend existiert fast kein Algorithmus, dessen Ausführungszeit bei unterschiedlichen Parametern immer gleich bliebe. Wer den Artikel »Dem Klang auf der Spur (5)« (64'er, Ausgabe 5/85, Seite 152 ff.) gelesen hat, weiß, welch grobe Differenzen bei Multiplikationen auftreten können.

Ersetzt (beziehungsweise unterstützt) man einen Algorithmus durch eine Multiplikationstabelle, fällt eine einheitlichere (und kürzere) Ausführungszeit an.

Für das Rechnen mit einzelnen Bits in einem Byte werden oft die Zweierpotenzen benötigt; es empfiehlt sich, diese als Tabelle anzulegen:

```
1000 -; Zweierpotenzen als Tabellle
1010 -; im DOS der Floppy 1541 ab $EFE9
1020 -; zu finden
1030 -; ZWEIPOT .BY 2↑0, 2↑1, 2↑2, 2↑3, 2↑4,
        2↑5, 2↑6, 2↑7
```

Folgende Unterroutine legt im Akkumulator den Wert 2↑A ab, wobei mit A der Inhalt des Akkumulators bei Aufruf der Routine gemeint ist:

```
10000 -;
10010 -; Subroutine zur Berechnung von
10020 -; 2↑A (Ergebnis kommt in den Akku)
10030 -;
10040 - TAX ; Akku in Indexregister
10050 - LDA ZWEIPOT,X ; aus Tabelle einlesen
10060 - RTS ; Das war's schon! Wer ein
        schnelleres und zugleich so einfaches
        Verfahren kennt, möge sich melden...
10070 - ZWEIPOT
        .BY 2↑0,2↑1,2↑2,2↑3,2↑4,2↑5,2↑6,2↑7
```

Wenn A größer als 7 ist, liefert das Programm falsche Werte. Sie können es noch erweitern, wenn Sie es für nötig halten.

b) Tabellen aus Fließkommawerten

Zu den zeitraubendsten Operationen gehört die Rechnung mit Fließkommazahlen. Daß diese selbst in Maschinenprogrammen lähmend wirkt, sehen Sie am HiRes-3-Befehl »FUNKT« (64'er, Ausgabe 3/85, Grafikkurs-Anwendung). Daher sollte man nur dann auf die Fließkommaroutinen zugreifen, wenn es unvermeidbar ist. Berechnen Sie soviele Werte wie möglich voraus, hierfür eignet sich der Direktmodus des Basic-Interpreter besonders gut! Wie Sie einen auf diese Weise berechneten Wert ins MFLPT-(Floating Point)Format umwandeln können, zeigt Ihnen der folgende Kasten.

> **Verfahren zur Umwandlung einer Zahl ins MFLPT-Format**
>
> 1. SMON (oder anderen Monitor) laden
> 2. Reset auslösen oder NEW eingeben
> 3. »XX = Fließkommazahl« eingeben, zum Beispiel »XX = 1.23456«
> 4. Monitor starten (SYS 49152)
> 5. »M 0805 0809« eingeben
>
> Sie sehen nun in den Adressen $0805 - $0809 die MFLPT-Darstellung der Zahl, mit der Sie die Variable XX belegt haben.

Damit wir uns unter Zuhilfenahme präziser Fachausdrücke und Abkürzungen verständigen können, sollten Sie den Abschnitt in »Assembler ist keine Alchimie« aufmerksam lesen, der sich mit Fließkommazahlen befaßt. Nach dem Studium dieses Abschnitts sollten Ihnen Begriffe wie »MFLPT«, »FAC« oder »ARG« geläufig sein.

Im Falle der Zahl 1.23456 erhalten wir als Ergebnis:

```
:0805 81 1E 06 0F E5...
```

Diese Werte legen wir folgendermaßen als Tabelle ab:

```
540 -BSPZAHL .BY $81, $1E, $06, $0F, $E5
```

Wie wir nun diese Zahl verarbeiten, zeigt Ihnen Listing 8. Das Makro (200 - 240) stützt sich auf die Interpreter-Routine MEMFAC, die eine Zahl (Adresse wird in Akku/Y-Register übergeben) vom Speicherformat MFLPT in den FAC als FLPT-Zahl schreibt und dabei die erforderliche MFLPT-→FLPT-Umwandlung durchführt.

In der Tabelle in Zeile 540 können Sie beliebige Fließkommawerte (sofern Sie diese wie angegeben berechnet haben) einsetzen, das Programm rechnet dann mit der jeweiligen Fließkommazahl, die ab BSPZAHL im MFLPT-Format steht.

Diese Zahl wird zunächst nur in den FAC geladen und der FAC wird dann ausgedruckt (270 - 290), dann wird die Zahl wieder geholt, die Wurzel berechnet und ausgegeben (310 - 350). Schließlich wird die Zahl wieder in den FAC geholt, der natürliche Logarithmus errechnet und auch ausgegeben (370 - 410).

Zur Routine FACOUT sind, außer daß sie den Inhalt des FAC ausgibt, noch zwei Bemerkungen zu machen:

1. Nach der Zahl wird noch ein CARRIAGE RETURN ausgegeben.
2. Nach dem Aufruf von FACOUT hat sich der Inhalt des FAC aufgrund mehrerer Divisionen durch Zehnerpotenzen verändert.

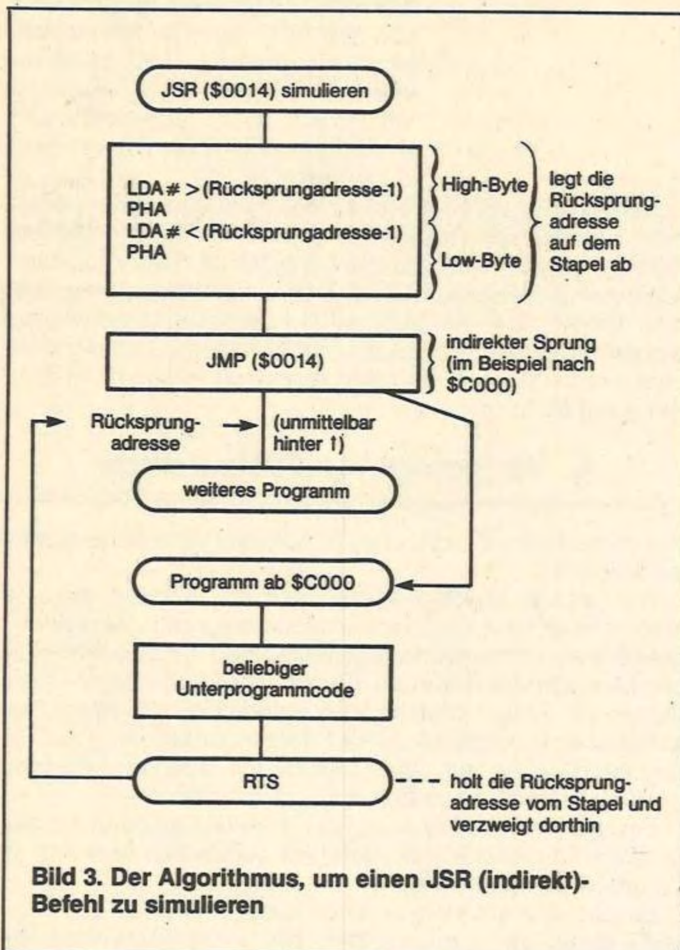


Bild 3. Der Algorithmus, um einen JSR (indirekt)-Befehl zu simulieren

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -; RECHNUNG MIT FLIESSKOMMAWERTEN
140  -;
150  -.GL MEMFAC = $BBA2
160  -.GL FACOUT = $AABC
170  -.GL SQRFAC = $BF71
180  -.GL LOGNAT = $B9EA
190  -;
200  -.MA HOLE (ADRESSE) ; MAKRO-DEF.
210  -          LDA #<(ADRESSE); HOLT MFLPT-ZAHL
220  -          LDY #>(ADRESSE); VON ADRESSE IN
230  -          JSR MEMFAC     ; DEN FAC
240  -.RT
250  -;
260  -;
270  -...HOLE (BSPZAHL)
280  -;
290  -          JSR FACOUT     ; AUSDRUCKEN
300  -;
310  -...HOLE (BSPZAHL)
320  -;
330  -          JSR SQRFAC     ; QUADRATWURZEL
340  -;
350  -          JSR FACOUT     ; AUSDRUCKEN
360  -;
370  -...HOLE (BSPZAHL)
380  -;
390  -          JSR LOGNAT     ; LOGARITHMUS NATURALIS
400  -;
410  -          JMP FACOUT     ; AUSDRUCKEN
500  -;
510  -; BEISPIELZAHL 1.23456
520  -; IM MFLPT-FORMAT
530  -;
540  -BSPZAHL   .BY $81,$1E,$06,$0F,$E5
550  -;
```

Listing 8. Fließkommazahlen in Assembler verarbeiten

Auf das Thema »Fließkommaarithmetik« geht Texteinschub 1 noch näher ein. Dort werden auch weitere Interpreter-Routinen vorgestellt.

c) Sprungtabelle

Beim Thema »Unterprogramme« wurde Ihnen eine Methode vorgestellt, um JSR (ind) zu simulieren. Diese erweist sich in Verbindung mit einer Tabelle, in der die Sprungadressen gespeichert sind, als sehr nützlich. So kann beispielsweise eine Parallele zum Basic-Befehl ON...GOSUB ZIEL1,ZIEL2.... geschaffen werden.

Ein Beispiel: Wenn der Basic-Interpreter auf einen Basic-Befehl trifft, holt er aus der Tabelle $A00C - $A09D die Adresse der zugehörigen Routine. Diese springt er dann durch Stapelmanipulation an.

Der SMON arbeitet genauso: Seine Sprungtabelle liegt im Bereich $C02B - $C06B.

Die Anwendung von Sprungtabellen werden wir noch ausführlich im folgenden Abschnitt d) sowie bei der Besprechung von Listing 11 behandeln.

d) Vergleichstabellen

Weder der SMON noch der Basic-Interpreter benutzen zum Suchen der zum jeweiligen Befehl gehörenden Routine eine Reihe von CMP-Abfragen mit BRANCH-Befehlen. Auch für die Vergleichswerte (in diesem Fall die Befehlswörter) gibt es eine Tabelle: Beim SMON liegt sie im Bereich $C00B - $C02A, beim Basic-Interpreter $A09E - $A327.

Sprung- und Vergleichstabellen sind in gleicher Befehlsfolge angeordnet; wird der Befehl an einer bestimmten Stelle in der Vergleichstabelle gefunden, erfolgt ein Sprung an die Adresse, die an gleicher Stelle in der Sprungtabelle steht.

So sehen die Befehls- und Vergleichstabellen im SMON aus:

Spalte Nr.	1	2	3	4	....
Befehl	/	#	$	%	....
Sprungadr. $	CADB	C920	C908	C91C	....

Die Sprungadressen sind wegen der Stapelmanipulation in der Tabelle ab $C02B um 1 dekrementiert gespeichert; in der Darstellung sehen Sie aber das tatsächliche Sprungziel.

Wir werden jetzt anhand des SMON die Verwendung einer Vergleichs-Sprungtabelle in Assembler erläutern.

Wenn wir die zum Befehl »#« gehörende Sprungadresse finden wollen, gehen wir folgendermaßen vor:

1. Wir suchen in Reihe 2 das #-Zeichen.
2. Wir gehen (in derselben Spalte) eine Reihe nach unten und finden dort die Sprungadresse ($C920).

Der Computer hat nicht die Möglichkeit, direkt eine Reihe weiter unten die Suche fortzusetzen. Er muß einen Umweg wählen und sich die Spalte merken. Ein Beispiel:

1. Der SMON sucht unter den Elementen aus Reihe 2 das »#«. In einem Zähler merkt er sich die Spalte, in der der Befehl gefunden wurde.
2. Nun sucht er in Reihe 3 in der Spalte, die im Zähler steht, die zugehörige Sprungadresse.

Wie ähnlich beide Suchvorgänge sind, erkennen Sie daran, daß jedesmal die Hauptschritte 1. und 2. vorkommen.

Nach so viel Theorie sehen wir uns nun um so ausführlicher die Routine im SMON an, die für die Steuerung der Vergleichs-Befehlstabelle verantwortlich ist. Dazu können Sie »D C303 C323« eingeben.

Bei Adresse $C303 steht im Akku der ASCII-Code des Kommandos, das der SMON ausführen soll (zum Beispiel $40, wenn ein M-Befehl eingegeben wurde).

C303 LDX #$20	32-1 Befehle müssen durchsucht werden. Weshalb »–1« erforderlich ist, liegt an der Schleifenstruktur und ist unbedeutend.
C305 CMP $C00A,X	Akku (enthält Befehl) mit X-tem Element der Befehlstabelle vergleichen; $C00A = Befehlstabelle -1, weil Adresse $C00A nie zum Vergleich herangezogen wird.
C308 BEQ $C30F	Vergleich positiv; im X-Register steht jetzt die Spalte.
C30A DEX	Zähler wird dekrementiert; es handelt sich hier um eine »Dekrementierschleife« (dieses Thema wird noch behandelt).
C308 BNE $C305	Wenn der Zähler noch nicht gleich 0 ist, folgt ein Sprung zum Schleifenbeginn.
C30D BEQ $C2D1	Wenn X=0, dann wurde die ganze Tabelle durchsucht, und der Befehl nicht gefunden! Deshalb wird in die SMON-Fehlerbehandlung gesprungen.
C30F JSR $C315	Diese Stelle wird von $C308 aus angesprungen; hier wiederum steht ein Aufruf des Unterprogramms ab $C315, das etwas weiter unten besprochen wird.
C312 JMP $C2D6	Nachdem nun der Befehl durch die Subroutine $C315 abgearbeitet wurde, folgt ein Sprung zur Eingabe des nächsten Befehls.

C315 TXA	Das ist sie, die Subroutine! Weil im X-Register die Nummer des Befehls (= Spalte in Tabelle) steht, kommt das X-Register ins Hauptrechenregister.
C316 ASL	Die Befehlsnummer wird mit 2 multipliziert...
C317 TAX	und kommt wieder ins X-Register. Die Multiplikation mit 2 ist erforderlich, weil in der Sprungtabelle ein Element doppelt so lang ist, wie in der Vergleichstabelle, nämlich 2 Byte. Die Sprungadressen belegen deshalb 2 Byte, weil sie aus Low- und High-Bytes bestehen.
C318 INX	Das X-Register wird um 1 erhöht, da das High-Byte eine Position hinter dem Low-Byte steht.
C319 LDA $C029, X	High-Byte wird gelesen. Die Sprungtabelle beginnt zwar 2 Byte nach $C029, aber weil es keine Spalte 0 gibt, muß der Speicherbedarf einer Sprungadresse (=2) abgezogen werden.
C31C PHA	Das High-Byte der Adresse wird auf den Stapel gelegt.
C31D DEX	-1, weil Low-Byte eine Adresse vor High-Byte steht.
C31E LDA $C029,X	Nun wird auch das Low-Byte der Adresse
C321 PHA	auf den Stapel geschoben.
C322 RTS	Der Befehl RTS wird hier zur Simulation von JMP (ind) verwendet. Auf dieses (unpraktische) Verfahren soll nicht weiter eingegangen werden (der 6510 kennt ja den indirekten Sprungbefehl). Wichtig ist für uns nur, daß jede SMON-Routine mit einem RTS abgeschlossen wird, dann erfolgt ein Rücksprung zur Adresse $C312.

Damit haben wir SMONs Schleife zum Suchen eines Befehls und dessen Routine durchleuchtet. Sofern Sie ein ROM-Listing zur Verfügung haben, können Sie sich zusätzlich die entsprechenden Stellen im Basic-Interpreter ansehen. Dieser aber benötigt wegen seiner unterschiedlich langen Befehle einen etwas komplizierteren Suchalgorithmus, was wiederum zu einer erheblich höheren Ausführungszeit beiträgt.

6. Vergleiche von Prüfsummen

Nun lernen wir ein besonders raffiniertes Vergleichsverfahren kennen:

Wie gesagt, benötigen Vergleiche mit Wörtern, die aus unterschiedlich vielen Zeichen bestehen, mehr Taktzyklen. Dies wäre nicht so, wenn wir alle Zeichen auf eine einheitliche Länge bringen würden. Genau dies tut der Basic-Interpreter: Bei Eingabe einer Zeile wandelt er alle Basic-Befehlswörter in Token um. Jedes Token vertritt einen Befehl und kann, da es nur ein Byte benötigt, schneller erkannt werden, als es bei mehreren Bytes möglich wäre.

Ein Nachteil ist jedoch der Speicherplatzaufwand; für die Umwandlung müssen die Befehle irgendwo im Speicher in Langform vorhanden sein.

Es gibt aber noch ein anderes Verfahren, einer Zeichenkette einen Wert zuzuweisen: Die Prüfsummenberechnung. Diese führen zum Beispiel die Eingabehilfen »Checksummer« und »MSE« durch: Aus 8 KByte Programmcode und 2 Byte Adresse errechnet der MSE eine 1 Byte lange Prüfsumme.

In Bild 4 sehen Sie einen sehr zuverlässigen Algorithmus zur Berechnung von Prüfsummen (insofern zuverlässig, als er sehr unterschiedliche Prüfsummen ermittelt). Listing 9 stellt ein Hilfsprogramm dar, das zu einer Eingabe die Prüfsumme nach dem Algorithmus aus Bild 4 errechnet.

In Listing 9 ist Ihnen eventuell die Routine NUMOUT nicht bekannt. Daher eine Kurzbeschreibung«: NUMOUT gibt eine positive Integerzahl, die im Akkumulator (High-Byte) und im X-Register (Low-Byte) übergeben wird, aus. NUMOUT wird zum Beispiel von der LIST-Routine bei der Ausgabe einer Zeilennummer aufgerufen.

Die Routine BASIN soll ebenfalls erklärt werden, da sie in allen folgenden Programmen verwendet werden wird. Wenn die Routine BASIN zum ersten Mal aufgerufen wird, erwartet das Betriebssystem eine Eingabe (normalerweise von Tastatur), die der Eingabe einer Basic-Ziele entspricht. Nach der Eingabe wird das erste eingebene Byte in den Akku geladen, jeder weitere Aufruf von BASIN holt das nächste Zeichen in den Akku. Wurden alle Bytes eingelesen, wird im Akku der Wert 13 ($OD, RETURN) übergeben. Danach führt ein weiterer Aufruf von BASIN zu erneuter Eingabe von Tastatur.

Ein großer Vorteil von Prüfsummen ist, daß die Vergleiche mit nur einem Byte, nämlich der Prüfsumme, durchgeführt werden müssen.

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG    LDA #<(TEXT1)
180  -          LDY #>(TEXT1)
190  -          JSR STROUT
200  -;
210  -          LDX #0
220  -SCHLEIFE1 JSR BASIN
230  -          CMP #13          ; 13 = RETURN
240  -          BEQ WEITER
250  -          STA STORE,X
260  -          INX
270  -          JMP SCHLEIFE1
280  -;
290  -WEITER    STX LAENGE
300  -          LDA #<(TEXT2)
310  -          LDY #>(TEXT2)
320  -          JSR STROUT
330  -          LDA #0
340  -; 0 = AUSGANGSWERT DER PRUEFSUMME
350  -          TAX              ; ZAEHLER = 0
360  -SCHLEIFE2 ROL              ; PRUEFSUMME * 2
370  -          EOR STORE,X
380  -          INX              ; ZAEHLER ERHOEHEN
390  -          CPX LAENGE
400  -          BNE SCHLEIFE2
410  -          CLC
420  -          ADC LAENGE       ; LAENGE ADDIEREN
430  -          TAX              ; PRUEFSUMME
440  -          LDA #0           ; AUSGEBEN
450  -          JSR NUMOUT
460  -          JMP ANFANG       ; NOCH EINMAL
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1     .BY 13
1040 -.TX "---------------------------------------"
1050 -.TX "EINGABE ?  "
1060 -.BY 0
1070 -;
1080 -TEXT2     .BY 13
1090 -.TX "PRUEFSUMME "
1100 -.BY 0
2000 -;
2010 -; ZWISCHENSPEICHER
2020 -;
2030 -LAENGE    .BY 0            ; ZWISCHENSPEICHER
2040 -STORE     .BY 0
2050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 9. Die Berechnung von Prüfsummen

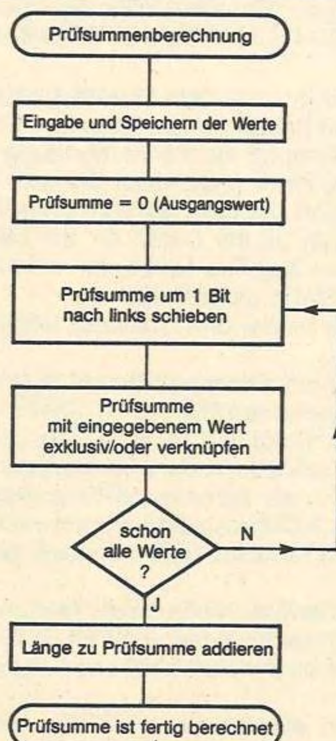


Bild 4. Das Flußdiagramm zur Prüfsummenberechnung

Wie man in den Genuß dieses Vorteils kommt, zeigt Listing 10. Wenn Sie den Namen eines Computers (C 64, VC 20, PC 128 oder AMIGA) eingeben, nennt das Programm den in diesem Computer installierten Mikroprozessor. Bei der Eingabe der Computernamen kann man aufgrund der Zeilen 230 und 240 beliebig viele Leerzeichen eingeben. Bei der Errechnung der Prüsummen mit Listing 9 dürfen allerdings keine eingegeben werden, da Listing 9 diese nicht überliest und somit ein falsches Ergebnis liefern würde.

Der Programmteil, der die Prüfsumme der Eingabe berechnet, ist mit Ausnahmen der Zeilen 230/240 aus Listing 9 übernommen worden. Nach Zeile 450 wird die ermittelte Prüfsumme mit der Tabelle »PRÜFSUMMEN« (Zeile 2060) verglichen.

Bei »WEITER2« (Zeile 620) steht im X-Register die Spalte, in der die Prüfsumme gefunden wurde. Listing 10 numeriert, im Gegensatz zum SMON die Spalten mit 0 (statt mit 1) beginnend. Außerdem wurde die Adressentabelle in »LOWTAB« (Tabelle der Low-Bytes) und »HIGHTAB« (High-Bytes) zerlegt, was die Programmierung stark erleichtert.

Wir würden zwar Spalten von 1 an numerieren, für den Computer ist es aber besser, mit Spalte 0 zu beginnen. Wenn im X-Register die Spalte (0: VC 20, 1: C 64, 2: PC 128, 3: AMIGA) steht, lesen die Zeilen 620/630 aus einer Tabelle die Adresse, ab der die ASCII-Darstellung des Prozessornamens zu finden ist. Weil jede der Tabellen »LOWTAB« und »HIGHTAB« gleich viele Elemente wie die Tabelle »PRUEFSUMMEN« hat, muß keine komplizierte Umwandlung über Multiplikation mit 2 oder ähnliches erfolgen wie beispielsweise beim SMON.

Auf eine akute Gefahr bei der Verwendung von Prüfsummen soll jetzt hingewiesen werden: die »Überschneidung von Prüfsummen«:

So wie unterschiedliche Basic-Zeilen beim Checksummer eine gleiche Prüfsumme haben können, sind Prüfsummen nie eindeutig.

Wenn Sie bei Listing 10 etwas herumprobieren, werden Sie sicher feststellen, daß auch eigentlich nicht vorgesehene Eingaben Wirkung zeigen. Dies liegt daran, daß diese Eingaben die gleiche Prüfsumme wie die Taste, »C64«, »PC 128« oder »AMIGA« haben. Daher sollte man immer darauf achten, daß sich die vorgesehenen Eingaben nicht in ihren Prüfsummen überschneiden (das heißt, die gleichen Prüfsummen haben). Wenn man dies aber beachtet, so ist das Arbeiten mit Prüfsummen, vor allem bei kleineren Datenmengen, eine angenehme Sache.

e) Beispielprogramm für Tabellen

Wenden wir uns jetzt einem etwas größeren (aber keineswegs komplizierteren) Programm zu. Es heißt schlicht und einfach »TABELLEN-BEISPIEL«, womit schon einiges über die Funktion ausgesagt ist: ein reines Beispielprogramm, das nicht den Anspruch erhebt, etwa als Anwendersoftware nützlich zu sein. In Listing 11 finden Sie den kommentierten Quelltext.

Zuerst soll die Bedienung des Programms erläutert werden. Gestartet wird »TABELLEN-BEISPIEL« durch »SYS 49152« und <RETURN>, worauf man sich in folgendem Menü befindet:

```
ZAHL IN ZAHLWORT WANDELN     (0)
BILDSCHIRMFARBE              (1)
RESET AUSLOESEN              (2)
PROGRAMMENDE UEBER RTS       (3)
BITTE AUSWAEHLEN!
```

Die Zahlen in Klammern sehen Sie nicht, diese zeigen nur die interne Numerierung der Menüpunkte an.

Der jeweils angewählte Menüpunkt (unmittelbar nach dem Start: 0) wird im Gegensatz zu den anderen revers hervorgehoben.

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG    LDA #<(TEXT1)
180  -         LDY #>(TEXT1)
190  -         JSR STROUT
200  -;
210  -         LDX #0
220  -SCHLEIFE1 JSR BASIN
230  -         CMP #" "       ; SPACE?
240  -         BEQ SCHLEIFE1  ; DANN UEBERLESEN
250  -         CMP #13        ; 13 = RETURN
260  -         BEQ WEITER1
270  -         STA STORE,X
280  -         INX
290  -         JMP SCHLEIFE1
300  -;
310  -WEITER1   STX LAENGE
320  -         LDA #<(TEXT2)
330  -         LDY #>(TEXT2)
340  -         JSR STROUT
350  -         LDA #0
360  -; 0 = AUSGANGSWERT DER PRUEFSUMME
370  -         TAX            ; ZAEHLER = 0
380  -SCHLEIFE2 ROL            ; PRUEFSUMME * 2
390  -         EOR STORE,X
400  -         INX            ; ZAEHLER ERHOEHEN
410  -         CPX LAENGE
420  -         BNE SCHLEIFE2
430  -         CLC
440  -         ADC LAENGE     ; LAENGE ADDIEREN
450  -; HIER STEHT DIE PRUEFSUMME IM AKKU
460  -;
470  -         LDX #0
480  -SCHLEIFE3 CMP PRUEFSUMMEN,X
490  -         BEQ WEITER2
500  -         INX
510  -         CPX #4
520  -         BNE SCHLEIFE3
530  -; PRUEFSUMME NICHT GEFUNDEN
540  -;
550  -         PLA
560  -         PLA
570  -         LDA #<(TEXT3)
580  -         LDY #>(TEXT3)
590  -         JSR STROUT
600  -         JSR ANFANG     ; VON VORNE
610  -;
620  -WEITER2   LDA LOWTAB,X   ; LOW-BYTE
630  -         LDY HIGHTAB,X  ; HIGH-BYTE
640  -         JSR STROUT
650  -         JMP ANFANG     ; NOCH EINMAL!
660  -;
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1      .BY 13
1040 -.TX "----------------------------------------"
1050 -.TX "COMPUTER : "
1060 -.BY 0
1070 -;
1080 -TEXT2      .BY 13
1090 -.TX "PROZESSOR: "
1100 -.BY 0
1110 -;
1120 -TEXT3      .TX "WEISS ICH NICHT!"
1130 -.BY 0
1140 -;
1150 -;
1160 -T6502      .TX "MOS 6502"
1170 -.BY 0
1180 -;
1190 -T6510      .TX "MOS 6510"
1200 -.BY 0
1210 -;
1220 -T8502      .TX "MOS 8502 & Z80"
1230 -.BY 0
1240 -;
1250 -T68000     .TX "MOTOROLA 68000"
1260 -.BY 0
1270 -;
2000 -;
2010 -; NUMERISCHE TABELLEN
2020 -;
2030 -LOWTAB     .BY <(T6502),<(T6510),<(T8502),<(T68000)
2040 -HIGHTAB    .BY >(T6502),>(T6510),>(T8502),>(T68000)
2050 -;
2060 -PRUEFSUMMEN .BY 228,83,149,136
2070 -; REIHENFOLGE: VC20,C64,PC128,AMIGA
3000 -;
3010 -; ZWISCHENSPEICHER
3020 -;
3030 -LAENGE     .BY 0              ; ZWISCHENSPEICHER
3040 -STORE      .BY 0
3050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 10. Eine Anwendung der Prüfsummenberechnung

Der angewählte Menüpunkt kommt durch Drücken von <F1>, <RETURN>, <←> oder <=> zur Ausführung.

Wollen Sie einen anderen Menüpunkt anwählen, drücken Sie einfach <Cursor - abwärts>, <D>, <F5> oder <+>, um den invertierten Bereich nach unten zu bewegen. Weiter nach oben gelangen Sie über <Cursor - aufwärts>, <U>, <F3> oder <->

Wenn Sie von »3« aus nach unten wollen, geht es wieder bei »0« los; von »0« nach oben führt auf Punkt »3«.

Auf Punkt »0« (Ausgangseinstellung) kommen Sie über <HOME>, <0> oder <@>.

Sicher würden Sie Ihre Programme auch gerne mit einem solch komfortablen Menü aufwerten. Wenn Sie die Beschreibung des Quelltextes gut durchlesen, wird dies keine Schwierigkeiten bereiten.

Nun zu den einzelnen Menüpunkten.

»2« (Reset auslösen) springt in die Reset-Routine ab $FCE2. »3« (Programmende über RTS) bewirkt einen Rücksprung ins Basic. Wenn Sie aber »TABELLEN-BEISPIEL« vom Hypra-Ass aus gestartet haben, finden Sie sich im »AUTONUMBER«-Modus wieder. Dies ist weder ein Fehler von »TABELLEN-BEISPIEL« noch von Hypra-Ass, sondern liegt daran, daß beide Programme eine bestimmte Adresse verwenden, die Hypra-Ass dann als Aufforderung zur automatischen Zeilennumerierung wertet. Am besten starten Sie »TABELLEN-BEISPIEL« nur vom normalen Basic aus.

Punkt »0« bittet Sie um Eingabe einer Zahl von 0 bis 9 und gibt zur eingegebenen Zahl das Zahlwort aus. Beispiel: Eingabe »0«, Ausgabe »NULL«.

Danach müssen Sie eine beliebige Taste drücken, um ins Hauptmenü zu kommen.

Punkt »1« schließlich bietet die Möglichkeit, die Hintergrundfarbe besonders elegant einzustellen: Sie geben einfach die Farbe als Wort ein, zum Beispiel SCHWARZ.

Folgende Eingaben sind vorgesehen:
SCHWARZ,WEISS,ROT,TUERKIS,VIOLETT,GRUEN,
BLAU,GELB,ORANGE,BRAUN,HELLROT,GRAU 1,
GRAU 2,HELLGRUEN,HELLBLAU,GRAU 3

Aufgrund der Überschneidung von Prüfsummen zeigen jedoch auch andere Eingaben Wirkung, zum Beispiel:
SCH,HYPRA ASS,PRINT,COMPUTER-GRAPHIK,
TAGESSCHAU

Nun wollen wir uns mit dem Quelltext befassen.

Ab Zeile 10000 finden Sie die Tabellen. Und weil unser Programm ein Beispiel für die Verwendung von Tabellen sein soll, sind es derer recht viele. Die wichtigsten davon sind jedoch analog der internen Numerierung der Menüpunkte aufgebaut, da sie Daten für die Menüsteuerung beinhalten. Diese Tabellen sind auch mit 0 - 3 numeriert und grafisch in Bild 6 dargestellt.

Sehen wir uns wieder den Quelltext, beginnend mit der ersten Zeile, an.

Auf die Symboldefinitionen (210 - 260) folgt die Initialisierung der Hauptschleife (280 - 310). Diese Initialisierung löscht Bildschirm (280) und Tastaturpuffer (290 - 300). Außerdem wird der aktuelle (= derzeit invers dargestellte) Menüpunkt (immer in der Adresse »MPT« enthalten) auf 0 gesetzt (310). Zeile 310 ist also dafür verantwortlich, daß nach dem Start über »SYS 49152« das Inversfeld ganz oben steht (auf Punkt 0).

Die Texte, die der Beschreibung der Menüpunkte dienen, werden in der Hauptschleife »HSCHLEIFE« (350 - 550) ausgegeben. Mit dieser wollen wir uns nun eingehend auseinandersetzen.

Zunächst wird die Tabelle »RVSTAB« gelöscht (350 - 400). Diese Tabelle enthält die Information, ob der erläuternde Text zu einem Menüpunkt invers ausgegeben wird. Wenn nein, so enthält das entsprechende Byte eine »0«, andernfalls eine »18« (= REVERS-ON-Code für Betriebssy-

stem). Das entsprechende Byte aus »RVSTAB« braucht nur vor dem Menüpunkt-Text ausgegeben werden (470 - 480). Die Zeilen 410 - 430 sorgen dafür, daß das Byte in »RVSTAB«, welches sich auf den aktuellen Menüpunkt bezieht, den RVS-ON-Code erhält.

In der Hauptschleife muß das X-Register in »XSAVE« gesichert werden, weil die Routine »STROUT« den Inhalt des X-Registers ändert.

Mit »TASTE« (610) beginnt dann die Tastaturabfrage im Menü. Die Routine »GET« holt ein Zeichen von Tastatur als ASCII-Code in den Akku. Wurde keine Taste gedrückt, erhält der Akku den Code 0. In diesem Fall wartet 620 auf eine neue Eingabe. Beachten Sie bitte, daß der Akku nach der Zeile 620 NIE den Wert 0 haben kann (dies wird sich bald als nützlich erweisen)!

Wurde nun eine Taste gedrückt, sucht »SCHLEIFE3« (630 - 680) in der Tabelle »TASTEN«, die im Quelltext ab Zeile 10210 steht, nach dem eingegebenen Zeichen (wird es nicht gefunden, erfolgt in 690 der Sprung zur neuen Eingabe).

```
100  -.BA $C000 ; START: SYS 49152
110  -;
120  -; *************************
130  -; *                       *
140  -; *  TABELLEN - BEISPIEL  *
150  -; *  ===================  *
160  -; *                       *
170  -; *  BY  FLORIAN MUELLER  *
180  -; *                       *
190  -; *************************
200  -;
210  -.GL STROUT = $AB1E
220  -.GL CURSORHOME = $E566
230  -.GL GET = $FFE4
240  -.GL BASIN = $FFCF
250  -.GL BASOUT = $FFD2
260  -.GL RESET = $FCE2 ; SOFTWARE-RESET
270  -;
280  -START      JSR $E544       ; = PRINT CHR$(147)
290  -           LDA #0          ; TASTATURPUFFER
300  -           STA 198         ; LOESCHEN
310  -           STA MPT
320  -; ↑ SETZT AKTUELLEN MENUEPUNKT AUF 0
330  -HSCHLEIFE JSR CURSORHOME
340  -; ↑ HSCHLEIFE = HAUPTSCHLEIFE
350  -           LDA #0
360  -           TAX
370  -SCHLEIFE1 STA RVSTAB,X
380  -           INX
390  -           CPX #4
400  -           BNE SCHLEIFE1
410  -           LDX MPT
420  -           LDA #18         ; 18 = REVERS EIN
430  -           STA RVSTAB,X
440  -           LDX #0
450  -; ↑ SCHLEIFENZAEHLER INITIALISIEREN
460  -SCHLEIFE2 STX XSAVE       ; X RETTEN
470  -           LDA RVSTAB,X
480  -           JSR BASOUT
490  -           LDA TEXTLO,X    ; ERKLAERUNG
500  -           LDY TEXTHI,X    ; ZUM MENUEPUNKT
510  -           JSR STROUT      ; AUSGEBEN
520  -           LDX XSAVE       ; X WIEDER HOLEN
530  -           INX
540  -           CPX #4
550  -           BNE SCHLEIFE2
560  -;
570  -;
580  -; HIER IST DAS MENUE BEREITS AUF
590  -; DEN BILDSCHIRM AUSGEGEBEN WORDEN.
600  -;
610  -TASTE      JSR GET         ; TASTATURABFRAGE
620  -           BEQ TASTE       ; WARTEN AUF TASTENDRUCK
630  -           LDX #0
640  -SCHLEIFE3 CMP TASTEN,X
650  -           BEQ WEITER1
660  -           INX
670  -           CPX #16
680  -           BNE SCHLEIFE3
690  -           JMP TASTE
700  -WEITER1    TXA
710  -           LSR             ; DIVIDIERT AKKU-
720  -           LSR             ; MULATOR DURCH 4
730  -           TAX
740  -           LDA SP1LO,X
750  -           STA SPRUNG
760  -           LDA SP1HI,X
770  -           STA SPRUNG+1
780  -;
790  -.EQ RUECKSPRUNG = HSCHLEIFE-1
800  -; ↑ LEGT RUECKSPRUNGADRESSE DES
810  -;   UNTERPROGRAMMS FEST.
820  -;
830  -           LDA #>(RUECKSPRUNG)
840  -           PHA
850  -           LDA #<(RUECKSPRUNG)
860  -           PHA
870  -           JMP (SPRUNG)
880  -;
890  -;
900  -HOME       LDX #0
910  -           STX MPT
920  -ENDE       RTS             ; ENDE DES UNTERPRG
930  -;
940  -DOWN       LDX MPT         ; MENUEPUNKT
950  -           INX             ; UM 1 ERHOEHEN
960  -           CPX #4          ; GROESSER ALS 3?
970  -           BEQ HOME        ; DANN =0
980  -           STX MPT         ; SONST UEBERNEHMEN
990  -           RTS             ; ZUR HAUPTSCHLEIFE
1000 -;
1010 -UP         LDX MPT         ; MENUEPUNKT
1020 -           DEX             ; DEKREMENTIEREN
1030 -           BPL ENDUP       ; > 0?
1040 -           LDX #3          ; NEIN, DANN =3
1050 -ENDUP      STX MPT         ; UND UEBERNEHMEN
1060 -           RTS             ; ZUR HAUPTSCHLEIFE
1070 -;
1080 -;
1090 -EXEC       PLA             ; STAPELMANIPULATION
1100 -           PLA
1110 -           LDX MPT
1120 -           LDA SP2LO,X
1130 -           STA SPRUNG
1140 -           LDA SP2HI,X
1150 -           STA SPRUNG+1
1160 -           JMP (SPRUNG)
1170 -;
1180 -;
1190 -;
1200 -ZAHLWORT   LDA #<(TZAHL)   ; AUFFORDERUNG
1210 -           LDY #>(TZAHL)   ; ZUR EINGABE
1220 -           JSR STROUT      ; AUSGEBEN
1230 -           JSR BASIN       ; HOLT ZEICHEN
1240 -           SEC             ; IN BINAERZAHL
1250 -           SBC #"0"        ; UMWANDELN
1260 -           TAX             ; INS X-REGISTER
1270 -;
1280 -; JETZT STEHT IM X-REGISTER
1290 -; DIE EINGEGEBENE ZAHL
1300 -;
1310 -           CMP #10         ; > 10?
1320 -           BCC ZAHLWORT1   ; NEIN=> WEITER
1330 -           JMP ZAHLWORT    ; NEUEINGABE
1340 -;
1350 -ZAHLWORT1 STX XSAVE       ; X RETTEN
1360 -           LDA #<(TWORT)   ; AUFFORDERUNG
1370 -           LDY #>(TWORT)   ; ZUR EINGABE
1380 -           JSR STROUT      ; AUSGEBEN
1390 -           LDX XSAVE       ; X WIEDER HOLEN
1400 -           LDA ZWLO,X      ; ADRESSE DES
1410 -           LDY ZWHI,X      ; ZAHLWORTES HOLEN
1420 -           JSR STROUT      ; UND Z.WORT DRUCKEN
1430 -;
1440 -WAIT       JSR GET         ; WARTET AUF
1450 -           BEQ WAIT        ; TASTENDRUCK
1460 -           JMP START       ; ZUM HAUPTMENUE
1470 -;
1480 -;
1490 -;
1500 -FARBE      LDA #<(TFARBE)
1510 -           LDY #>(TFARBE)
1520 -           JSR STROUT
1530 -           LDX #0
1540 -FARBE1     JSR BASIN       ; HOLT EINGABE
1550 -           CMP #" "        ; SPACE ?
1560 -           BEQ FARBE1      ; JA=>UEBERLESEN
1570 -           CMP #13         ; ENDE DER EINGABE?
1580 -           BEQ FARBE2      ; JA, DANN WEITER
1590 -           STA FARBWORT,X ; EINGABE SPEICHERN
1600 -           INX             ; ZAEHLER ERHOEHEN
1610 -           JMP FARBE1      ; ZUR SCHLEIFE
1620 -FARBE2     STX 2           ; LAENGE MERKEN
1630 -           LDX #0
1640 -           TXA
1650 -FARBE3     ROL
1660 -           EOR FARBWORT,X
1670 -           INX
1680 -           CPX 2           ; SCHON FERTIG?
1690 -           BNE FARBE3      ; NEIN,ZUR SCHLEIFE
1700 -           CLC             ; LAENGE
1710 -           ADC 2           ; ADDIEREN
1720 -;
1730 -; HIER STEHT IM AKKU DIE PRUEFSUMME
1740 -;
1750 -           LDX #0
1760 -FARBE4     CMP PRUEFSUMMEN,X
1770 -           BEQ FARBE5      ; GEFUNDEN
1780 -           INX
1790 -           CPX #16
1800 -           BNE FARBE4
1810 -           JMP FARBE       ; NEUE EINGABE
1820 -FARBE5     STX 53280       ; BILDSCHIRM-
```

Listing 11. »Tabellen-Beispiel«, ein Beispiel zur Verwendung von Tabellen

```
1830 -          STX 53281       ; FARBE SETZEN
1840 -          JMP START       ; ZUM MENUE
1850 -;
10000-;
10010-; TABELLEN
10020-; ========
10030-;
10040-; TEXTE:
10050-;
10060-PUNKT0     .TX "ZAHL IN ZAHLWORT UMWANDELN"
10070-.BY 13,13,0
10080-;
10090-PUNKT1     .TX "BILDSCHIRMFARBE"
10100-.BY 13,13,0
10110-;
10120-PUNKT2     .TX "RESET AUSLOESEN"
10130-.BY 13,13,0
10140-;
10150-PUNKT3     .TX "PROGRAMMENDE UEBER RTS"
10160-.BY 13,13,13
10170-.TX "BITTE AUSWAEHLEN !"
10180-.BY 0
10190-;
10200-;
10210-TASTEN     .BY 133,13,"←","="; 133=F1,13=RETURN
10220-.BY 19,"0","@",0 ; 19=HOME,0=DUMMY
10230-.BY 17,"D",135,"+" ; 17=CRSR DOWN,135=F5
10240-.BY 145,"U",134,"-" ; 145=CRSR UP,134=F3
10250-;
10260-;
10270-TZAHL      .BY 147          ; CLEAR HOME
10280-.TX "ZAHL (0-9) ? "
10290-.BY 0
10300-;
10310-TWORT      .TX " IN WORTEN : "
10320-.BY 0
10330-;
10340-;
10350-; ZAHLWOERTER (0-9)
10360-;
10370-;
10380-NULL       .TX "NULL"
10390-.BY 0
10400-;
10410-EINS       .TX "EINS"
10420-.BY 0
10430-;
10440-ZWEI       .TX "ZWEI"
10450-.BY 0
10460-;
10470-DREI       .TX "DREI"
10480-.BY 0
10490-;
10500-VIER       .TX "VIER"
10510-.BY 0
10520-;
10530-FUENF      .TX "FUENF"
10540-.BY 0
10550-;
10560-SECHS      .TX "SECHS"
10570-.BY 0
10580-;
10590-SIEBEN     .TX "SIEBEN"
10600-.BY 0
10610-;
10620-ACHT       .TX "ACHT"
10630-.BY 0
10640-;
10650-NEUN       .TX "NEUN"
10660-.BY 0
10670-;
10680-;
10690-TFARBE     .BY 147          ; CLEAR HOME
10700-.TX "WELCHE FARBE ? "
10710-.BY 0
10720-;
10730-;
10740-RVSTAB     .BY 0,0,0,0      ; 4 BYTES RESERVIEREN
10750-;
10760-;
10770-; ZAHLEN:
10780-;
10790-; ADRESSEN DER TEXTE, DIE DIE
10800-; MENUEPUNKTE BESCHREIBEN
10810-;
10820-TEXTLO     .BY <(PUNKT0),<(PUNKT1)
10830-.BY <(PUNKT2),<(PUNKT3)
10840-;
10850-TEXTHI     .BY >(PUNKT0),>(PUNKT1)
10860-.BY >(PUNKT2),>(PUNKT3)
10870-;
10880-;
10890-; ADRESSEN DER ZAHLWOERTER
10900-;
10910-ZWLO       .BY <(NULL),<(EINS),<(ZWEI),<(DREI)
10920-.BY <(VIER),<(FUENF),<(SECHS),<(SIEBEN)
10930-.BY <(ACHT),<(NEUN)
10940-;
10950-ZWHI       .BY >(NULL),>(EINS),>(ZWEI),>(DREI)
10960-.BY >(VIER),>(FUENF),>(SECHS),>(SIEBEN)
10970-.BY >(ACHT),>(NEUN)
10980-;
10990-;
11000-; ADRESSEN DER UNTERROUTINEN
11010-; FUER DIE MENUESTEUERUNG
11020-;
11030-SP1LO      .BY <(EXEC),<(HOME),<(DOWN),<(UP)
11040-;
11050-SP1HI      .BY >(EXEC),>(HOME),>(DOWN),>(UP)
11060-;
11070-;
11080-; ADRESSEN DER EINZELNEN
11090-; MENUEPUNKTE
11100-;
11110-SP2LO      .BY <(ZAHLWORT),<(FARBE)
11120-.BY <(RESET),<(ENDE) ; BEI ENDE STEHT
11130-SP2HI      .BY >(ZAHLWORT),>(FARBE)
11140-.BY >(RESET),>(ENDE) ; EIN RTS-BEFEHL
11150-;
11160-; PRUEFSUMMEN DER FARB-WOERTER
11170-;
11180-PRUEFSUMMEN .BY 41,158,137,212,159,101
11190-.BY 3,2,33,69,201,116,113,121,127,114
11200-;
11210-;
11220-; ZWISCHENSPEICHER
11230-;
11240-MPT        .BY 0            ; 1 BYTE RESERVIEREN
11250-XSAVE      .BY 0
11260-SPRUNG     .WO 0            ; 2 BYTES FREIHALTEN
11270-FARBWORT   .BY 0
11280-; ↑ AB 'FARBWORT' WIRD DIE EINGABE
11290-;   DER FARB-BEZEICHNUNG ABGELEGT.

READY.
```

Listing 11. »Tabellen-Beispiel«, ein Beispiel zur Verwendung von Tabellen

Diese Tabelle »TASTEN« enthält alle vorgesehenen Tastendrücke zur Menüsteuerung, die in 4er-Blockweise angeordnet sind (Bild 5). Nach der Suchschleife steht im X-Register die Position der gedrückten Taste innerhalb der Tabelle »TASTEN« (zum Beispiel 0 = F1 gedrückt, 4 = HOME gedrückt). Diese Position wird - ohne Berücksichtigung des Divisions-Restes - durch 4 dividiert (700 - 730), um festzuhalten, von welchem Tastenblock eine Taste gedrückt wurde. Dadurch ist eindeutig bestimmt, welche Befehlsgruppe aufgerufen werden muß.

Steht nach 730 im X-Register 0, wurde eine der ersten vier in »TASTEN« enthaltenen Tasten gedrückt, die die Ausführung des aktuellen Menüpunktes veranlassen (Zeile 10210 und Bild 5). Ist X=1, so wurde eine Taste aus Zeile 10220 gedrückt. In 10220 steht als letztes Byte eine 0. Diese dient, da für die Funktion »Inversfeld in HOME-Position« nur drei Tastendrücke vorgesehen wurden, zum Auffüllen auf vier Tasten. 0 kann hier bedenkenlos als Dummy (Füllbyte ohne wirkliche Bedeutung) stehen, da der Akku aufgrund von 620 nie den Wert 0 annehmen wird. Beinhaltet X nach der Division durch 4 den Wert 2, wird das Inversfeld nach unten bewegt, ist X=3, dann nach oben. Dies können Sie sich an Bild 6 veranschaulichen.

An den Zeilen 740 - 870 sehen wir die Verwendung einer Sprungtabelle. Unsere Sprungtabelle ist »SP1LO/SP1HI«. »SP1LO« beinhaltet die Low-, »SP1HI« die High-Bytes der anzuspringenden Routinen. In den Vektor »SPRUNG« wird die Zieladresse geschrieben (740 - 770).

Die Zuweisungszeile 790 errechnet die Rücksprungadresse des aufzurufenden Unterprogramms. Bei einem RTS soll nämlich zur »HSCHLEIFE« gesprungen werden.

Diese Rücksprungadresse »RUECKSPRUNG« wird auf den Stapel gelegt (830 - 860), zuletzt erfolgt der indirekte Sprung (870). Die über die soeben beschriebene Simulation von JSR (ind) angesprungenen Routinen finden Sie ab Zeile 900. Es wird einfach der aktuelle Menüpunkt »MPT« entsprechend dem Tastendruck geändert, dann wird zur »HSCHLEIFE« gesprungen, die auch die Tabelle »RVSTAB« entsprechend anpaßt.

»EXEC« (1090) holt die Rücksprungadresse vom Stapel (1090 - 1100), da diese Routine nicht als Unterprogramm behandelt werden soll.

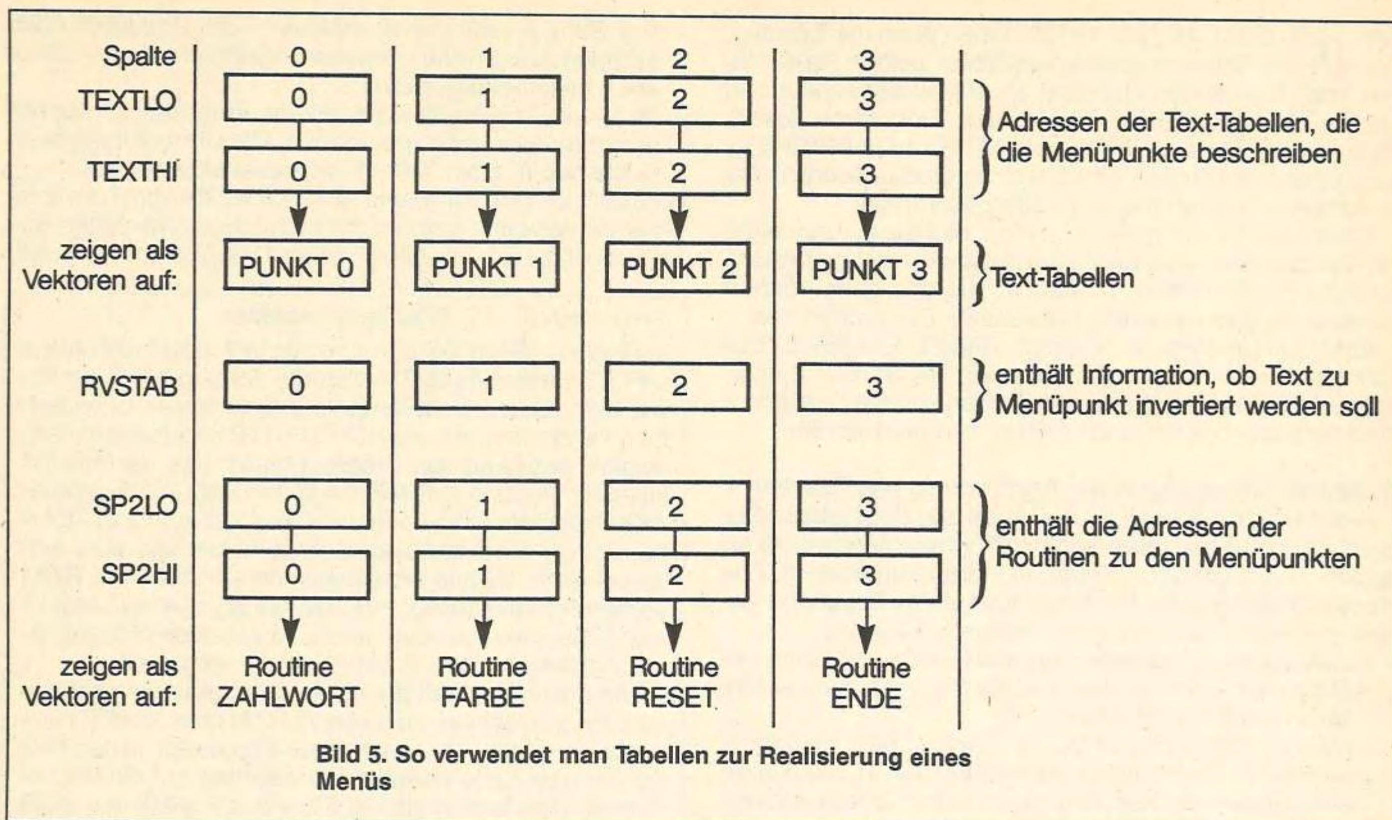


Bild 5. So verwendet man Tabellen zur Realisierung eines Menüs

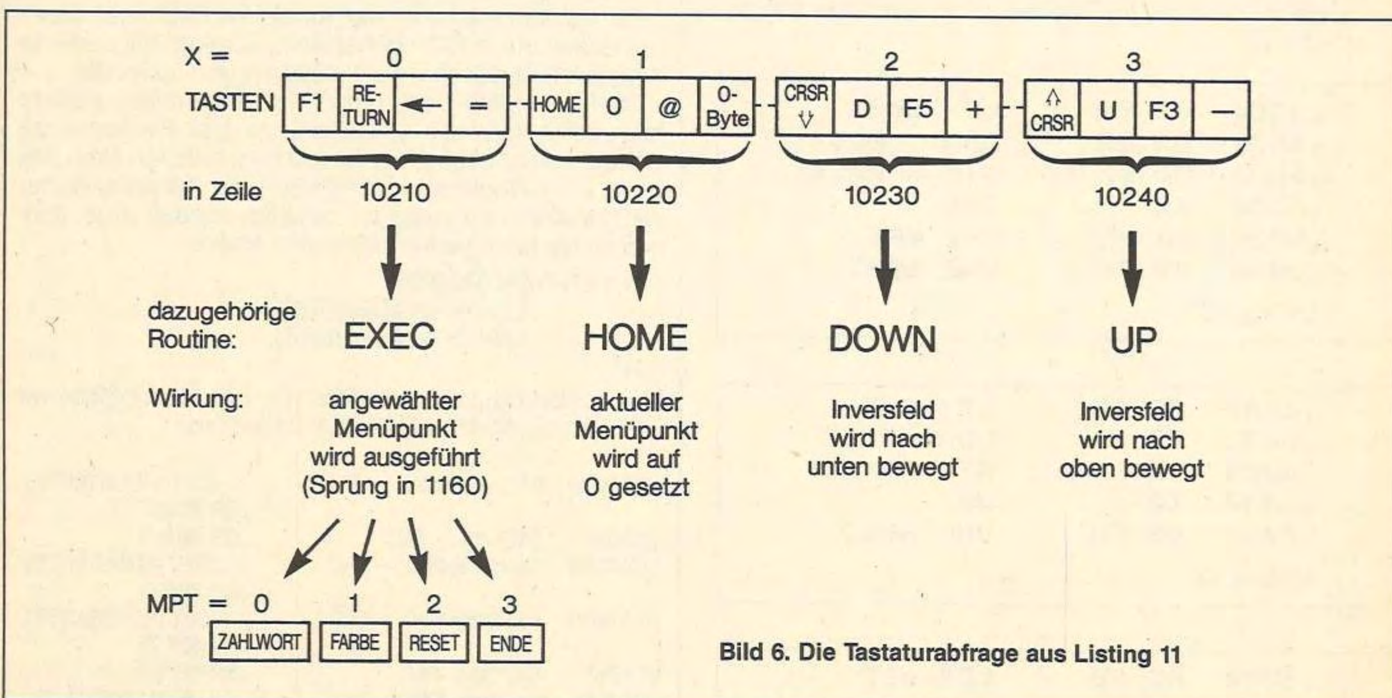


Bild 6. Die Tastaturabfrage aus Listing 11

Die Zeile 1110 holt den angeforderten Menüpunkt ins X-Register. Dann wird aus »SP2LO/SP2HI« die Adresse der zum Menüpunkt gehörenden Routine geholt und diese über einen gewöhnlichen indirekten Sprung aufgerufen (1160).

Als Routine zu »2« wird einfach die RESET-Routine des Betriebssystems angesprungen, für »3« eignet sich jeder RTS-Befehl, also auch der bei »ENDE« (920).

»ZAHLWORT«, die Routine zu 0, holt eine Zahl als ASCII-Code (1230) und wandelt sie in einen numerischen Wert um (1240 - 1250), indem der ASCII-Code von 0 abgezogen wird. Das Ergebnis landet im X-Register (1260). Ob auch eine Zahl eingegeben wurde, prüfen die Zeilen 1310 - 1330. Bei »ZAHLWORT« (1350) wird das Resultat der Subtraktion in »XSAVE« gesichert, der Text »IN WORTEN« ausgegeben und das X-Register wieder geholt.

Die Tabelle »ZWLO/ZWHI« enthält die Adressen, ab denen die Texte der Zahlwörter als ASCII-Code stehen. Aus »ZWLO/ZWHI« wird dann diese Adresse geholt (1400 - 1410) und der dort stehende Text ausgegeben (1420). Danach erwartet das Programm noch einen Tastendruck (1440–1450), bevor ins Hauptmenü verzweigt wird (1460).

Als letzte Routine wird »FARBE« besprochen (1500–1850): Hierzu ist jedoch aufgrund der Ähnlichkeit zu Listing 10 nicht viel zu erläutern. Bei 1820 steht im X-Register der Code der eingegebenen Farbe (= Position der Prüfsumme innerhalb der Tabelle »PRUEFSUMMEN«). Dieser muß nur noch in die entsprechenden VIC-Register geschrieben wer-

den (1820–1830). Ab Zeile 10000 stehen dann die Tabellen. Wenn Sie die Tabellen angesehen haben, sollten Sie durchaus noch einmal den Quelltext bis 10000 betrachten und die hier endende Beschreibung des Programms lesen. Denn wenn Sie das Programm »TABELLEN-BEISPIEL« ganz verstanden haben, sind Sie einen großen Schritt in der Assembler-Programmierung weitergekommen!

Ich könnte mir übrigens vorstellen, daß Sie in Ihren eigenen Programmen jetzt auch eine Menüsteuerung wie die in »TABELLEN-BEISPIEL« einbauen; wie das geht, können Sie dem Programm »TABELLEN-BEISPIEL« entnehmen.

Eine Anmerkung ist wichtig: »TABELLEN-BEISPIEL« kann noch weiter verbessert werden. Sie werden sehen, daß viele Stellen noch optimiert werden können. Insbesondere der Speicherplatzbedarf kann verringert werden.

f) Weitere Anregungen zur Anwendung von Tabellen

Auch die bisherigen Erläuterungen und das Beispielprogramm können die Kreativität des Programmierers nicht ersetzen, sondern nur die Programmierung erleichtern. Aus diesem Grund möchte ich Ihnen noch einige Beispiele nennen, wie sich Tabellen sinnvoll verwerten lassen.

- Ein Anwenderprogramm, das aus Menüs und Untermenüs besteht, sollte in einer Tabelle die Adressen der Menüs/Untermenüs speichern.
- Spiele müssen oft viele Sprite-Bewegungen, die immer gleich sind, durchführen. Es empfiehlt sich, die Sprite-Bewegungen als Koordinaten in einer Tabelle abzulegen. Diese Methode wurde beispielsweise im Artikel »Tanzende Sprites« im Sonderheft 30 auf Seite 148 beschrieben.

```
,6000  A2 00     LDX #00
,6002  B5 02     LDA   02,X
,6004  9D 00 6F  STA 6F00,X
,6007  E8        INX
,6008  E0 FE     CPX #FE
,600A  D0 F6     BNE 6002
```

Listing 12

```
,6000  A2 FE     LDX #FE
,6002  B5 01     LDA #01,X
,6004  9D FF 6E  STA 6EFF,X
,6007  CA        DEX
,6008  D0 F8     BNE 6002
```

Listing 13

```
,6000  A2 FE     LDX #FE
,6002  BD FF 6E  LDA 6EFF,X
,6005  95 01     STA #01,X
,6007  CA        DEX
,6008  D0 F8     BNE 6002
```

Listing 14

```
,6000  A2 34     LDX #34
,6002  B5 16     LDA   16,X
,6004  9D 00 6F  STA 6F00,X
,6007  CA        DEX
,6008  10 F8     BPL 6002
```

Listing 15

- Bei Software-Interfaces müssen viele Umrechnungen erfolgen. Durch eine Umwandlungstabelle können diese stark beschleunigt werden.
- Naturwissenschaftlich orientierte Programme müssen verschiedene Maße umrechnen. Die Umrechnungswerte können in einer Tabelle untergebracht werden.

Dies soll nur eine Anregung sein. Ich wüßte aber kein komplexes Programm, das sich nicht durch den gezielten Einsatz von Tabellen vereinfachen und beschleunigen ließe.

Texteinschub #1: Fließkommazahlen

Im Text wurde ein Verfahren vorgestellt, um eine Zahl ins MFLPT-Format (MELPT=Memory floating point) umzuwandeln. Das 5 Byte lange Ergebnis dieser Umwandlung kann man dann als KONSTANTE handhaben. Konstanten sind feste, vorausberechnete Werte, die man mit Hilfe der Routine »MEMFAC« in den FAC (Fließkomma-AKKU) kopieren kann. Für viele Werte ist es jedoch überflüssig, die Umwandlung durchzuführen und eine entsprechende Tabelle anzulegen, da sie schon im ROM vorhanden sind. Im vorangehenden Kurs »Assembler ist keine Alchimie« wurden solche Konstanten mitsamt ihren Adressen schon in einer Tabelle vorgestellt.

Um mit Konstanten (für die Rechenroutinen macht es keinen Unterschied, ob diese im RAM oder im ROM stehen) zu rechnen, kann man diese wie gesagt, in den FAC kopieren und alle weiteren Operationen auf diesen beziehen. Dies war in Listing 8 bei den Funktionen SQR und LOGNAT ausreichend.

Oft möchte man aber den Inhalt des FAC nicht mit einer Funktion wie SQR behandeln, sondern mit anderen Konstanten addieren, multiplizieren und so weiter.

Dafür möchte ich Ihnen im folgenden weitere Interpreter-Routinen vorstellen, die das Rechnen mit Konstanten ermöglichen. Da fast immer in den Akku das Low-, ins Y-Register das High-Byte der Adresse, ab der die Konstante abgelegt ist, geladen werden muß, definieren wir noch vorher folgendes Makro:

```
-.MA LDAY(ADRESSE)
-         LDA #<(ADRESSE)
-         LDY #>(ADRESSE)
-.RT
```

Nun zu den Routinen, bei deren Parameterübergabe wir uns auf das Makro LDAY stützen wollen:

ADDMEM	FAC+Konstante → FAC	...LDAY (KONSTANTE) JSR $B867
ADD0,5	FAC+0.5 → FAC	JSR $B849
SUBMEM	Konstante-FAC → FAC	...LDAY (KONSTANTE) JSR $B850
MULMEM	Konstante*FAC → FAC	...LDAY (KONSTANTE) JSR $BA28
MULT10	FAC*10 → FAC	JSR $BAE2
DIVMEM	Konstante/FAC → FAC	...LDAY (KONSTANTE) JSR $BB0F
DIVS10	FAC/10 → FAC	JSR $BAFE
CMPMEM	vergleicht Konstante mit FAC FAC<Konstante: Akku=$FF FAC=Konstante: Akku=$00 FAC>Konstante: Akku=$01	...LDAY (KONSTANTE) JSR $BC5B
POTMEM	Konstante ↑FAC → FAC	...LDAY (KONSTANTE) JSR $BF78
POTE	e↑FAC → FAC	JSR $BFED
MEMFAC	holt Konstante in FAC	...LDAY (KONSTANTE) JSR $BBA2
FACMEM	FAC ab Konstante als MFLPT-Zahl ablegen	LDX #<(KONSTANTE) LDY #>(KONSTANTE) JSR $BBD7
FACOUT	gibt FAC aus	JSR $AABC

```
,6000  A2 34     LDX #34
,6002  BD 00 6F  LDA 6F00,X
,6005  95 16     STA   16,X
,6007  CA        DEX
,6008  10 F8     BPL 6002
```
Listing 16

```
,6000  A2 FF     LDX #FF
,6002  BD 00 01  LDA 0100,X
,6005  9D 00 6F  STA 6F00,X
,6008  CA        DEX
,6009  D0 F7     BNE 6002
,600B  AD 00 01  LDA 0100
,600E  8D 00 6F  STA 6F00
,6011  BA        TSX
,6012  8E 00 70  STX 7000
```
Listing 17

```
,6000  A2 FF     LDX #FF
,6002  BD 00 6F  LDA 6F00,X
,6005  9D 00 01  STA 0100,X
,6008  CA        DEX
,6009  D0 F7     BNE 6002
,600B  AD 00 6F  LDA 6F00
,600E  8D 00 01  STA 0100
,6011  AE 00 70  LDX 7000
,6014  9A        TXS
```
Listing 18

7. Die Initialisierung

»Initialisierung« nennt man eine Routine, die vor einem Programmteil (meist einer Schleife) steht und diese vorbereitet. Die Initialisierung wird nur einmal, eine Schleife aber mehrfach durchlaufen. Deshalb bringt es einen Geschwindigkeitszuwachs, wenn die Initialisierung der Schleife Arbeit abnimmt.

Ein Beispiel: Wenn ein Basic-Programm mit »RUN« gestartet wird, werden alle Variablen gelöscht, Files geschlossen und die Adressen, ab denen die Variablen abgelegt werden dürfen, errechnet. Dies ist die Initialisierung der Interpreterschleife. Dann wird Byte für Byte des Basic-Programms eingelesen und bearbeitet.

Muß im gerade übersetzten Befehl ein Sprung (GOTO 500 oder ähnliches) durchgeführt werden, kostet dies bekanntlich viel Zeit, wenn das Sprungziel am Ende eines langen Programms steht. Dies ist darauf zurückzuführen, daß der Interpreter, beginnend mit der ersten Zeile, das ganze Programm nach der Sprungzeile durchsucht, bis er sie gefunden hat.

Diese Berechnung der Adressen wird bei jedem »GOTO« oder »GOSUB« neu durchgeführt.

Viel besser und schneller wäre folgende Vorgehensweise: Bei »RUN« wird zunächst eine Tabelle angelegt, in der die Adressen aller Zeilen enthalten sind. Diese Tabelle könnte zum Beispiel als Array definiert werden. Folgt nun ein Sprung, kann aus der Tabelle die Adresse der Zeile im Speicher geholt werden.

Damit haben wir noch ein wesentliches Merkmal der Initialisierungsroutinen gefunden: Die Initialisierung kann Tabellen anlegen, die von der Hauptschleife verarbeitet werden.

Aber nicht nur Tabellen können generiert werden, auch die Berechnung von Flags ist sinnvoll. So merkt sich die »LOAD/VERIFY«-Routine ($FFD5), ob ein Verifizieren oder Laden gewünscht wird. Die Ladeschleife liest dann ein Zeichen von der Floppy oder der Datasette ein und entscheidet erst anschließend, ob das Byte im Speicher abgelegt oder mit dem Speicher verglichen werden soll.

Halten wir also fest, daß Initialisierungsroutinen Schleifen entlasten können.

8. Die Nutzung der Zeropage

In jedem Assembler-Lehrbuch werden die Vorteile der Zeropage-Adressierung gepriesen. Speicherplatzersparnis und hohe Verarbeitungsgeschwindigkeit sind nicht die einzigen Vorzüge; die indirekt-indizierte Adressierung kann nur auf Zeropage-Adressen zugreifen, nicht auf absolute 16-Bit-Adressen. Damit wird der Leser bereits allein gelassen. Er erfährt nicht, welche Adressen in der Zeropage für die Praxis geeignet sind. Das wird nun nachgeholt.

Fast die ganze Zeropage wird durch Basic-Interpreter und Betriebssystem belegt. Deshalb führen bestimmte Werte in Zeropage-Adressen oft zum Absturz oder sonstigem Fehlverhalten des Computers.

Wie dies im einzelnen aussieht, erfahren Sie im Artikel »Memory Map mit Wandervorschlägen«, der im Sonderheft 7(1986) erschien. Nicht nur in Zweifelsfällen stellt dieser Artikel das optimale Nachschlagewerk dar.

Ich möchte Ihnen nun zeigen, welche Adressen Sie als (Zwischen-)Speicher ohne Schwierigkeiten verwenden können, beziehungsweise was Sie bei Verwendung von Zeropage-Adressen beachten müssen.

a) Adressen, die problemlos verwendet werden können

Auf die Adressen $02 und $FB - $FE wird weder vom Basic-Interpreter noch vom Betriebssystem zugegriffen. Lediglich bei Initialisierung der Arbeitsspeicher (Reset) werden Sie auf 0 gesetzt.

Für die Praxis heißt das, daß Ihnen die genannten Adressen völlig zur Verfügung stehen.

b) Adressen, die in keiner Weise verwendet werden sollten

Von anderen Adressen hingegen müssen wir unsere Finger lassen. Diese haben entweder elementare Funktionen für Betriebssystem oder CPU, oder werden von beiden dauernd geändert, so daß die Datensicherheit in Frage gestellt ist. Genauer soll hier nicht unterschieden werden.

Belassen Sie die Adressen $00 und $01 unverändert, da sie (siehe Memory Map) für die CPU wichtige Informationen beinhalten und außerdem einige Bits nur durch externe Vorgänge geändert werden.

Das Betriebssystem und der Basic-Interpreter beanspruchen alle bislang ungenannten Adressen.

Von Bildschirmeditor und Tastaturabfrage werden die Adressen $C6 - $F6 beeinflußt. Die Adressen $90 - $C2 dienen der Ein-/Ausgabe-Steuerung mit Peripheriegeräten und der Verwaltung offener Files. Einzige Ausnahme: $A0 - $A2 (interne Uhr). Wenn ein Maschinenprogrammm in ein Basic-Programm eingebaut ist, sind die Adressen $03 - $56 sowie $73 - $8B tabu.

c) Bedingt einsetzbar

Der Vektor $C3/$C4 wird durch <RUN/STOP-RESTORE>, Reset oder »LOAD« beeinflußt. Ansonsten kann mit $C3/$C4 frei verfahren werden.

Ganz Vorsichtige können diesen Vektor auf seinen Ausgangswert $FD30 setzen, sobald das Programm die Adressen $C3/$C4 nicht mehr für eigene Zwecke benötigt.

```
,6000  A9 D2   LDA #D2
,6002  85 14   STA  14
,6004  A9 3F   LDA #3F
,6006  85 15   STA  15
,6008  A0 00   LDY #00
,600A  B1 14   LDA  (14),Y
,600C  49 FF   EOR #FF
,600E  91 14   STA  (14),Y
,6010  E6 14   INC  14
,6012  D0 02   BNE 6016
,6014  E6 15   INC  15
,6016  A5 14   LDA  14
,6018  C9 60   CMP #60
,601A  A5 15   LDA  15
,601C  E9 47   SBC #47
,601E  90 EA   BCC 600A
```

Listing 19

```
,6000  A9 5F   LDA #5F
,6002  85 14   STA  14
,6004  A9 47   LDA #47
,6006  85 15   STA  15
,6008  A0 00   LDY #00
,600A  B1 14   LDA  (14),Y
,600C  49 FF   EOR #FF
,600E  91 14   STA  (14),Y
,6010  A5 14   LDA  14
,6012  D0 02   BNE 6016
,6014  C6 15   DEC  15
,6016  C6 14   DEC  14
,6018  A5 14   LDA  14
,601A  C9 D2   CMP #D2
,601C  A5 15   LDA  15
,601E  E9 3F   SBC #3F
,6020  B0 E8   BCS 600A
```

Listing 20

```
,6000  A9 00   LDA #00
,6002  85 14   STA  14
,6004  A9 20   LDA #20
,6006  85 15   STA  15
,6008  A0 00   LDY #00
,600A  B1 14   LDA  (14),Y
,600C  49 FF   EOR #FF
,600E  91 14   STA  (14),Y
,6010  C8      INY
,6011  D0 F7   BNE 600A
,6013  E6 15   INC  15
,6015  A5 15   LDA  15
,6017  C9 40   CMP #40
,6019  D0 EF   BNE 600A
```

Listing 21

d) Adressen, die unter Verzicht auf Kassettenbetrieb verwendet werden können

Die folgenden Adressen können verwendet werden, wenn Sie nicht auf RS232 oder Datasette zugreifen.

$9E/$9F, $A5-$A7, $A9-$AB, $B0-$B6, $F7-$FA

Bei anderen Adressen, die sich auf den RS232- oder Kassettenbetrieb beziehen, ist Vorsicht angebracht.

e) Geeignete Zwischenspeicher

Die Adressen $22-$2A und $57-$60 sind sogenannte »verschieden genutzte Arbeitsbereiche«. Sie werden vom Basic-Interpreter vor allem bei arithmetischen Operationen als Zwischenspeicher verwendet. Als solche Zwischenspeicher können wir sie auch verwenden. Sobald allerdings bestimmte Interpreterroutinen aufgerufen werden, können die Inhalte dieser Adressen verlorengehen. Eine längerfristige Aufbewahrung von Daten in diesen Adressen ist zwar nicht möglich, andererseits können wir aber durch Schreibzugriffe auf diese Adressen das Betriebssystem oder den Basic-Interpreter nicht stören.

Zu sagen wäre noch, daß die Adressen $57 - $60 den wichtigen Routinen BLTUC und UMULT (siehe »Assembler ist keine Alchimie«) als Zwischenspeicher dienen.

```
,6000  A9 00   LDA #00
,6002  85 14   STA  14
,6004  A8      TAY
,6005  A9 20   LDA #20
,6007  85 15   STA  15
,6009  AA      TAX
,600A  B1 14   LDA  (14),Y
,600C  49 FF   EOR #FF
,600E  91 14   STA  (14),Y
,6010  C8      INY
,6011  D0 F7   BNE 600A
,6013  E6 15   INC  15
,6015  CA      DEX
,6016  D0 F2   BNE 600A
```

Listing 22

f) Zeropage kopieren

Zum Abschluß dieses Abschnittes über die Nutzung der Zeropage möchte ich Ihnen noch einen kleinen Trick verraten, der von einigen professionellen Programmen angewandt wird.

Wir sichern die Zeropage-Inhalte in einem anderen Bereich, zum Beispiel von $6F00 an.

Dann können wir viele Adressen in der Zeropage nutzen, sofern wir keine Interpreter- oder Betriebssystemroutine aufrufen. Danach schreiben wir die Zeropage wieder von der Kopie, zum Beispiel von $6F00, zurück und können wie gewöhnlich fortfahren.

Die Adressen 0 und 1 kopieren wir nicht, weil diese nach wie vor für solche Zwecke nutzlos sind. Ebenso könnten wir nur einzelne Bereiche kopieren (zum Beispiel die Zeiger für Basic-Programme $16 - $4A). Dann dürfen wir aber auch nur diesen Bereich verändern.

Wenn wir nun den Bereich $02 - $FF kopieren, stehen uns folgende Adressen zur Verfügung:

$03-$06, $14-$86, $71-$8A, $C3/$C4, $FB-$FF

Diese Adressen können Sie nur so lange verwenden, bis eine Routine des Betriebssystems oder Basic-Interpreters aufgerufen wird. Davor muß die alte Zeropage zurückgeschrieben werden.

Da Sie auf diese Weise viel Speicherplatz in der Zeropage gewonnen haben, ist es sogar möglich, eine Tabelle aus

Geschwindigkeitsgründen in diese zu verlegen. Damit steigt auch der Wert der indiziert-indirekten Adressierung erheblich.

Dennoch ist der Speicherplatz in der Zeropage begrenzt. Überlegen Sie sich also, auf welche Werte besonders schnell zugegriffen werden muß und schreiben Sie vorzugsweise diese in die Zeropage.

9. Schleifenprogrammierung

Zunächst befassen wir uns mit Schleifen, die maximal 256mal durchlaufen werden (die Zahlenangaben in allen Listings sind Hex-Zahlen).

Typ a: Schleifen mit maximal 256 Durchläufen

Da 256 verschiedene Zahlen mit einem 8-Bit-Prozessor dargestellt werden können, verwendet man hier das X- (oder Y-) Register als Schleifenzähler. In Listing 12 sehen Sie die einfachste Form einer Schleife, die die Zeropage-Adressen $02 - $FF nach $6F00 kopiert.

Da der Schleifenzähler X in Listing 12 inkrementiert wird, haben wir es mit einer INKREMENTIERSCHLEIFE zu tun. Nach dem Inkrementieren (»6007 INX«) wird durch »6008 CPX #FE« überprüft, ob die Schleife beendet werden kann. Eine eingehendere Beschreibung des Programmablaufs erübrigt sich.

Für Schleifen des Typs a (maximal 256 Durchläufe) ist es aber meist vorteilhaft, eine DEKREMENTIERSCHLEIFE zu verwenden. Wie eine solche Schleife programmiert wird, sehen wir an Listing 13.

Listing 13 unterscheidet sich in der Wirkung nicht von Listing 12, obwohl man dies nicht unbedingt auf den ersten Blick erkennt. Deshalb soll dieses Listing näher besprochen werden. In Zeile 6000 erhält das X-Register den Inhalt $FE. Durch »6002 LDA 01,X« wird damit das letzte Byte der Zeropage, nämlich $FF, zuerst gelesen und nach $6FFD

```
70   -.BA $C000
80   -.LI 1,3,0
90   -;
100  -; ******************************
110  -; *    QUELLTEXTE (HYPRA-ASS)   *
120  -; *    =====================    *
130  -; *                             *
140  -; * FUER VERSCHIEDENE SCHLEIFEN *
150  -; *                             *
160  -; * 28.08.85 BY FLORIAN MUELLER *
170  -; *                             *
180  -; ******************************
190  -;
200  -;
210  -; QUELLTEXT ZU LISTING 1
220  -; =====================
230  -;
240  -.EQ ANFANGSADRESSE = $02
250  -.EQ ENDADRESSE = $FF
260  -.EQ ZIELBEREICH = $6F00
270  -;
280  -          LDX #0
290  -SCHLEIFE1 LDA ANFANGSADRESSE,X
300  -          STA ZIELBEREICH,X
310  -          INX
320  -          CPX #(ENDADRESSE+1-ANFANGSADRESSE)
330  -          BNE SCHLEIFE1
340  -;
350  -;
360  -; QUELLTEXT ZU LISTING 2
370  -; =====================
380  -;
390  -.EQ ANFANGSADRESSE = $02
400  -.EQ ENDADRESSE = $FF
410  -.EQ ZIELBEREICH = $6F00
420  -;
430  -          LDX #(ENDADRESSE+1-ANFANGSADRESSE)
440  -SCHLEIFE2 LDA ANFANGSADRESSE-1,X
450  -          STA ZIELBEREICH-1,X
460  -          DEX             ; DEKREMENTIERBEFEHL
470  -          BNE SCHLEIFE2
480  -;
490  -;
500  -; QUELLTEXT ZU LISTING 4
510  -; =====================
520  -;
530  -.EQ ANFANGSADRESSE = $16
540  -.EQ ENDADRESSE = $4A
550  -.EQ ZIELBEREICH = $6F00
560  -;
570  -          LDX #(ENDADRESSE-ANFANGSADRESSE)
580  -SCHLEIFE3 LDA ANFANGSADRESSE,X
590  -          STA ZIELBEREICH,X
600  -          DEX
610  -          BPL SCHLEIFE3  ; PRUEFT N-FLAG
620  -;
630  -;
640  -; QUELLTEXT ZU LISTING 8
650  -; =====================
660  -;
670  -.EQ ANFANGSADRESSE = $3FD2
680  -.EQ ENDADRESSE = $475F
690  -.EQ ZAEHLER = $14
700  -;
710  -          LDA #<(ANFANGSADRESSE)
720  -          STA ZAEHLER
730  -          LDA #>(ANFANGSADRESSE)
740  -          STA ZAEHLER+1
750  -          LDY #0
760  -SCHLEIFE4 LDA (ZAEHLER),Y
770  -          EOR #$FF
780  -          STA (ZAEHLER),Y
790  -          INC ZAEHLER
800  -          BNE WEITER
810  -          INC ZAEHLER+1
820  -WEITER    LDA ZAEHLER
830  -          CMP #<(ENDADRESSE+1)
840  -          LDA ZAEHLER+1
850  -          SBC #>(ENDADRESSE+1)
860  -          BCC SCHLEIFE4
870  -;
880  -;
890  -; QUELLTEXT ZU LISTING 10
900  -; =====================
910  -;
920  -.EQ ANFANGSADRESSE = $2000
930  -.EQ ENDADRESSE  = $3FFF
940  -.EQ ZAEHLER = $14
950  -;
960  -          LDA #<(ANFANGSADRESSE)
970  -          STA ZAEHLER
980  -          LDA #>(ANFANGSADRESSE)
990  -          STA ZAEHLER+1
1000 -          LDY #0
1010 -SCHLEIFE5 LDA (ZAEHLER),Y
1020 -          EOR #$FF
1030 -          STA (ZAEHLER),Y
1040 -          INY
1050 -          BNE SCHLEIFE5
1060 -          INC ZAEHLER+1
1070 -          LDA ZAEHLER+1
1080 -          CMP #>(ENDADRESSE+1)
1090 -          BNE SCHLEIFE5
1100 -;
1110 -;
1120 -; QUELLTEXT ZU EINER SCHLEIFE,
1130 -; DIE DEN BEREICH $3FD2-$47D1
1140 -; KOMPLEMENTIERT
1150 -;
1160 -.EQ ANFANGSADRESSE = $3FD2
1170 -.EQ ENDADRESSE = $47D1
1180 -.EQ ZAEHLER = $14
1190 -;
1200 -          LDA #<(ANFANGSADRESSE)
1210 -          STA ZAEHLER
1220 -          LDA #>(ANFANGSADRESSE)
1230 -          STA ZAEHLER+1
1240 -          LDX #>(ENDADRESSE+1-ANFANGSADRESSE)
1250 -          LDY #0
1260 -SCHLEIFE6 LDA (ZAEHLER),Y
1270 -          EOR #$FF
1280 -          STA (ZAEHLER),Y
1290 -          INY
1300 -          BNE SCHLEIFE6
1310 -          INC ZAEHLER+1
1320 -          DEX
1330 -          BNE SCHLEIFE6
1340 -;
1350 -; ENDE VON LISTING 12
```

Listing 23

```
,6000  A0 00     LDY #00
,6002  B9 00 20  LDA 2000,Y
,6005  49 FF     EOR #FF
,6007  99 00 20  STA 2000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```

Listing 24

```
,6000  A0 00     LDY #00
,6002  B9 00 40  LDA 4000,Y
,6005  49 FF     EOR #FF
,6007  99 00 40  STA 4000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```

Listing 25

```
,6000  A9 00     LDA #00
,6002  8D 13 60  STA 6013
,6005  8D 18 60  STA 6018
,6008  A9 20     LDA #20
,600A  8D 14 60  STA 6014
,600D  8D 19 60  STA 6019
,6010  A0 00     LDY #00
,6012  B9 FF FF  LDA FFFF,Y
,6015  49 FF     EOR #FF
,6017  99 FF FF  STA FFFF,Y
,601A  C8        INY
,601B  D0 F5     BNE 6012
,601D  EE 14 60  INC 6014
,6020  EE 19 60  INC 6019
,6023  AD 19 60  LDA 6019
,6026  C9 40     CMP #40
,6028  D0 E8     BNE 6012
```

Listing 26

geschrieben. Dann wird X dekrementiert. Ist X noch nicht 0, so wird die Schleife erneut durchlaufen.

Der niedrigste X-Wert innerhalb der Schleife ist folglich 1; aufgrund von »6002 LDA 01,X« ist $02 die niedrigste Zeropage-Adresse, die kopiert wird. In Listing 12 ist 0 der niedrigste X-Wert. Die niedrigste Adresse aufgrund von »6002 LDA 02,X« ist also auch $02 (stimmt auffällig). Warum $FF die höchste kopierte Zeropage-Adresse ist, können Sie nun selbst den Listings 12 und 13 entnehmen.

Listing 14 ist eine Dekrementierschleife, die die Kopie der Zeropage wieder von $6F00 nach $02 zurückholt.

Der Vorteil von Dekrementierschleifen beim Typ a ist, daß zum Erkennen der Abbruchbedingung (X=0) kein Vergleichsbefehl erforderlich ist, weil nach dem DEX-Befehl automatisch das Z-Flag gesetzt wird, wenn X Null wird.

Das Entfallen des Vergleichsbefehls »CPX #« bringt eine Ersparnis von 2 Byte Speicherplatz sowie insgesamt 508 Taktzyklen Rechenzeit. Da jedoch bei $6F00 eine Seitenüberschreitung (eine Seite entspricht 256 Byte) vorliegt, schrumpft der Zeitgewinn auf 254 Taktzyklen (dies ließe sich aber vermeiden, indem wir die Zeropage nach $6F01 kopieren, womit durch »6004 STA $6F00,X« keine Seitenüberschreitung auftreten würde).

Nun wollen wir noch einen Sonderfall behandeln:

Dekrementierschleifen vom Typ a, bei denen der Ausgangswert für X kleiner als 129 ist.

In Listing 15 sehen Sie eine Schleife, die den Bereich $16 - $4A nach $6F00 kopiert, Listing 16 schreibt die Werte von $6F00 zurück nach $16. Selbstverständlich hätten wir das

```
,6000  A9 D2     LDA #D2
,6002  8D 11 60  STA 6011
,6005  8D 16 60  STA 6016
,6008  A9 3F     LDA #3F
,600A  8D 12 60  STA 6012
,600D  8D 17 60  STA 6017
,6010  AD 00 00  LDA 0000
,6013  49 FF     EOR #FF
,6015  8D 00 00  STA 0000
,6018  EE 11 60  INC 6011
,601B  EE 16 60  INC 6016
,601E  D0 06     BNE 6026
,6020  EE 12 60  INC 6012
,6023  EE 17 60  INC 6017
,6026  AD 11 60  LDA 6011
,6029  C9 60     CMP #60
,602B  AD 12 60  LDA 6012
,602E  E9 47     SBC #47
,6030  90 DE     BCC 6010
```

Listing 27

```
,6000  A2 00     LDX #00
,6002  8E 11 60  STX 6011
,6005  8E 14 60  STX 6014
,6008  A2 A0     LDX #A0
,600A  8E 12 60  STX 6012
,600D  8E 15 60  STX 6015
,6010  AE 00 00  LDX 0000
,6013  8E 00 00  STX 0000
,6016  EE 11 60  INC 6011
,6019  EE 14 60  INC 6014
,601C  D0 F2     BNE 6010
,601E  EE 12 60  INC 6012
,6021  EE 15 60  INC 6015
,6024  AE 12 60  LDX 6012
,6027  E0 C0     CPX #C0
,6029  D0 E5     BNE 6010
```

Listing 28

```
80   -.BA $6000
90   -.LI 1,3,0
100  -;
110  -; HYPRA-ASS-QUELLTEXT ZU EINER
120  -; SELBSTMODIFIZIERENDEN SCHLEIFE
130  -; (ARBEITET WIE LISTING 5)
140  -;
150  -; 1985 BY FLORIAN MUELLER
160  -;
170  -;
180  -.GL START = $A000
190  -.GL ENDE  = $BFFF
200  -;
210  -           LDX #<(START)
220  -           STX MOD1+1
230  -           STX MOD2+1
240  -           LDX #>(START)
250  -           STX MOD1+2
260  -           STX MOD2+2
270  -MOD1       LDX $FFFF
280  -MOD2       STX $FFFF
290  -           INC MOD1+1
300  -           INC MOD2+1
310  -           BNE MOD1
320  -           INC MOD1+2
330  -           INC MOD2+2
340  -           LDX MOD1+2
350  -           CPX #>(ENDE+1)
360  -           BNE MOD1
```

Listing 29

Problem auch so lösen können wie in Listing 13. Wir wollen aber noch eine andere Konstruktion von Dekrementierschleifen kennenlernen, die in diesem Sonderfall möglich ist. Besprechen wir also Listing 15.

Bei 6000 wird ins X-Register die Zahl geladen, die man zu $16 addieren muß, um $4A zu erhalten. Dadurch wird zunächst bei 6002 die Adresse $4A gelesen und nach $6F34 geschrieben. Bei 6007 wird dekrementiert. Neu ist der Verzweigungsbefehl: es wird das N-Flag überprüft. Ist X = $FF, wird das N-Flag gesetzt und »6008 BPL 6002« beendet die Schleife. Der niedrigste Wert von X, der innerhalb der Schleife vorkommt, ist demnach $00.

Der BPL-Befehl funktioniert nur, wenn der Ausgangswert von X <129 ist. Andernfalls wäre nämlich nach dem Dekrementieren X>127 und damit das N-Flag gesetzt. Dies aber hätte zur Folge, daß die Schleife nur einmal durchlaufen würde.

Zur soeben behandelten Schleifenkonstruktion sind noch zwei Dinge zu sagen; erstens, daß sie nur in diesem Sonderfall (X<129) möglich ist, und zweitens, daß sie nicht effektiver als eine Lösung wie in Listing 13 ist.

Allgemeine Gültigkeit hat aber folgende Regel für Schleifen vom Typ a:

> Bei Schleifen vom Typ a ist Dekrementieren effektiver als Inkrementieren, solange die Durchlaufzahl nicht 255 überschreitet.
> Bei 256 Durchläufen erweist sich Inkrementieren oft als besser.

An Listing 17 sehen wir ein Beispiel für den letzten Satz der Regel. Listing 17 kopiert die letzten 256 Speicherplätze des Stapels ($0100 - $01FF) und den Stapelzeiger nach $6F00 - $7000. Listing 18 schreibt den Stapel wieder zurück.

Die Dekrementierschleife (6000 - 600A) kopiert nun den Bereich $0101 - $01FF, $0100 wird nicht übertragen. Dies geschieht in 600B - 600F. Eine andere Möglichkeit wäre ein zeitraubender CPX #FF-Befehl nach »6008 DEX«.

6011 - 6013 sichert schließlich noch das SP-Register.

Hier ist in der Tat eine Inkrementierschleife besser. Ändern wir Listing 17 also in Listing 17a:

```
        LDX #00
- LOOP  LDA 0100,X
-       STA 6F00,X
-       INX          ;(!!)
-       BNE LOOP
-       TSX
-       STX 7000
```

Analog ergibt sich Listing 18:

```
        LDX #00
- LOOP  LDA 6F00,X
-       STA 0100,X
-       INX          ;(!!)
-       BNE LOOP
-       LDX 7000
-       TXS
```

In den Listings 17a und 18a habe ich diejenigen Befehle, die sich in der symbolischen Darstellung nicht von den Listings 17 und 18 unterscheiden, mit einem »–« markiert.

Typ b: Schleifen mit mehr als 256 Durchläufen

Während Schleifen des Typs a meist so schnell abgearbeitet werden, daß man es gar nicht bemerkt, dauern Typ-b-Schleifen oft eine oder mehrere Sekunden.

Deswegen wollen wir hier versuchen, den Zeitbedarf von Typ-b-Schleifen zu verringern.

Unsere erste Typ-b-Schleife (Listing 19) soll den Bereich von $3FD2 bis $475F invertieren (= EOR #FF-verknüpfen, aus jeder 1 wird eine 0 und umgekehrt). Da hierfür ein 8-Bit-Indexregister nicht ausreicht, benötigen wir einen 16-Bit-Zähler, nämlich $14/$15. Dieser soll immer die Adresse beinhalten, die invertiert wird. In diesen Zähler schreibt die Initialisierung der Schleife den Startwert $3FD2 (siehe $6000 - $6007).

Da es beim 6510 keine indirekte Adressierung für LDA/STA gibt, sondern nur die indirekt-indizierte oder indiziert-indirekte, müssen wir auf eine dieser Adressierungen ausweichen und den Index auf 0 setzen (»6008 LDY #00«).

Bei $600A beginnt die Schleife: der Wert wird eingelesen, mit $FF EOR-verknüpft und zurückgeschrieben. Nun erhöhen wir den 16-Bit-Zähler $14/$15 (6010 - 6015). Dann wird überprüft, ob die nächste Adresse schon mit der ersten Adresse nach der Endadresse ($475F), also $4760, übereinstimmt (siehe $6016 - $601C). Dieser 16-Bit-Vergleich wurde bereits im SMON vorgestellt. Bei $601E wird schließlich die Schleife beendet, falls die Abbruchbedingung (C=1) erfüllt ist.

Listing 20 ist eine Dekrementierschleife, die sich in der Wirkung nicht von Listing 19 unterscheidet. Da das Dekrementieren einer 16-Bit-Adresse beim 6510 langsamer und speicherplatzaufwendiger ist als das Inkrementieren, ist Listing 20 weniger effektiv als Listing 19.

Grundsätzlich können Sie an den Listings 19 und 20 sehen, wie man eine Typ-b-Schleife programmiert. Diese arbeitet jedoch nicht besonders schnell. Der Grund ist, daß der Bereich von $3FD2 - $475F nicht restlos in ganze Seiten (256-Byte-Blöcke) aufgeteilt werden kann. Daher sollte

man sich immer überlegen, ob die Schleifendurchlaufzahl nicht auf ganze 256-Byte-Blöcke »aufgerundet« werden kann. In unserem Fall würde dies heißen, daß mit einer schnelleren Schleife der exakt 8 x 256 Byte lange Bereich $3FD2 - $47D1 invertiert wird, anstelle des »ungeraden« Bereichs $3FD2 - $475F. An einfacheren Zahlen wollen wir nun eine solche Schleife für ganze Seiten programmieren. Der 32 x 256 Byte umfassende Bereich von $2000 bis $3FFF (einschließlich) soll invertiert werden. Mit einer solcher Routine könnte das gerade sichtbare Bild bei Hi-Eddi invertiert werden.

Die einfachste Form finden Sie in Listing 21. Zuerst wird die Anfangsadresse in $14/$15 abgelegt. Ins Y-Register kommt der Wert 0. Dann wird der Wert invertiert und das Y-Register, der Low-Zähler, erhöht. Ist der Wert noch nicht 0, wird die Schleife neu durchlaufen. Andernfalls wurde gerade eine Seite abgearbeitet. Der High-Zähler ($15) wird erhöht. Ist der Inhalt des High-Zählers = $40, wird die Schleife abgebrochen. Zu bemerken ist, daß während der Schleife die Adresse $14 unverändert 0 bleibt. Die Adresse, die invertiert wird, ergibt sich folgendermaßen:

(Y+Inhalt von $14)+256*(Inhalt von $15)

Da wir auf die Adresse über das Prozessor-Register Y Einfluß nehmen können und die Adresse $14 nicht verändert werden muß, ist die Verarbeitungsgeschwindigkeit gegenüber der »Normalform« (Listing 20) gestiegen. Das High-Byte müssen wir aber weiterhin in $15 belassen. Neu führen wir nun den High-Zähler X ein. Im X-Register merken wir uns, wie viele Seiten invertiert werden. Diesen Wert verwenden wir als Dekrementier-Zähler. In unserem Fall werden $20 Seiten invertiert. Weil $20 zufälligerweise auch das High-Byte der Anfangsadresse ($2000) ist, wird dieser Wert in Listing 22 nur einmal (6005) in den Akku geladen und dann bei 6009 ins X-Register übertragen.

Beachten Sie bitte, daß in Listing 22 die Befehle »6004 TAY« und »6009 TAX« nur bei den Werten dieses Beispiels verwendet werden können. In der Regel sind eigene »LDX #«- oder »LDY #«-Befehle erforderlich. Wenn wir zum Beispiel den Bereich $3FD2 - $47D1 invertieren wollen, muß die Initialisierung so aussehen:

```
LDA #D2  Low-Byte der ersten Adresse
STA 14
LDY #00  Index-Register
LDA #3F  High-Byte der ersten Adresse
STA 15
LDX #08  High-Zähler
```

... Schleife wie ab $600C in Listing 22

Damit hätten wir eine Schleife, die den Bereich #3FD2 - $475F (siehe Listings 19 und 20) invertiert und wesentlich schneller als die Listings 19 und 20 arbeitet. Da wir aber »aufgerundet« haben, wird zusätzlich der Bereich $4760 - $47D1 invertiert, obwohl wir das gar nicht wollen. Es gibt nun mehrere Möglichkeiten, dies zu verhindern:

1. Wir verwenden die Schleife aus Listing 19, müssen aber eine deutlich höhere Arbeitsdauer hinnehmen.

2. Wir verwenden die Schleife aus Listing 22 mit obiger Initialisierung. Dann invertiert eine Typ-a-Schleife den Restbereich $4760 - $47D1 ein weiteres Mal. Damit wären - eine Besonderheit der EOR #FF-Verknüpfung - im Restbereich die alten Inhalte wiederhergestellt. Diese Lösung eignet sich aber (fast) nur bei dieser logischen Verknüpfung und hilft bei den meisten anderen Typ-b-Schleifen nicht weiter.

3. Dies dürfte wohl die beste Lösung sein: Wir schreiben eine »gemischte« Schleife, die aus einer Typ-a-Schleife und einer Typ-b-Schleife besteht. Dieses Verfahren ist immer (!) möglich und wird von der BLTUC-Routine ($A3BF) des Basic-Interpreters angewandt. Diese Verschiebe-Routine zerlegt den Bereich, der verschoben werden soll, in einen Bereich, der aus 256-Byte-Blöcken besteht und in einen Restbereich. Beide Bereiche werden dann getrennt verschoben.

Folgendermaßen sieht die optimale Invertierroutine für den Bereich $3FD2 - $475F aus:

a) Der exakt 7 Seiten umfassende Bereich 3FD2 - $46D1 wird mit einer Typ-b-Schleife wie in Listing 22 komplementiert.

b) Der Restbereich $46D2 - $475F wird mit einer Typ-a-Schleife wie in Listing 13 komplementiert.

Wir haben nun viele verschiedene Schleifenkonstruktionen in Theorie und Praxis behandelt. Was uns noch fehlt, sind Formeln, nach denen Sie die einzelnen Parameter (zum Beispiel den Startwert für X in einer Dekrementier-Schleife vom Typ a) errechnen können. Als Zusammenfassung finden Sie in Form von Listing 23 ein Hypra-Ass-Assemblerlisting zu mehreren Schleifenkonstruktionen. An den Quelltext-Ausdrücken können Sie sehen, wie einzelne Parameter errechnet werden können.

Merke: Sofern es der Programmablauf zuläßt, sollten Sie Inkrementierschleifen verwenden.

Bei Verschiebeschleifen ist aber oft eine Dekrementierschleife erforderlich.

Noch etwas zum Schleifen-Inhalt: Wenn mehrere Schleifen einen gleichen Innenteil haben (zum Beispiel einen Invertierbefehl), definieren Sie diesen unbedingt als Makro und nicht als Unterprogramm! JSRs sollten Sie nur beim Aufruf von ROM-Routinen verwenden.

Damit wäre das Thema »Schleifen« erst einmal abgeschlossen. Im nächsten Abschnitt (über Selbstmodifikation) werden wir uns aber wieder mit Schleifen auseinandersetzen.

10. Selbstmodifikation

Bevor wir uns mit dieser Programmiertechnik beschäftigen, die zwar nicht strukturiert, aber sehr trickreich ist, soll der Begriff geklärt werden.

Unter Modifikation versteht man »eine Änderung, Anpassung«. Wenn Sie bei einem Spiel einen der vielen POKE-Befehle, die im 64'er-Magazin schon vorgestellt wurden, eingeben, so wird dadurch das Spiel »modifiziert«. Die Änderung ist zum Beispiel eine Erhöhung der Spielfigurenanzahl.

Selbstmodifikation bedeutet, daß ein Programm sich selbst programmgesteuert verändert. Dies wäre der Fall, wenn im Spielprogramm eine Passage stünde, die den POKE ausführt.

Auf simulierten Direktmodus wurde im 64'er-Magazin schon mehrfach eingegangen, unter anderem in der »Memory Map mit Wandervorschlägen«.

Wir werden uns an dieser Stelle ausschließlich mit der Selbstmodifikation von Maschinenprogrammen befassen. Als erstes Beispiel nehmen wir Listing 24.

Es handelt sich um eine selbstmodifizierende Schleife, die den Bereich $2000 - $3FFF komplementiert.

TRACEn Sie doch einmal Listing 24 mit dem SMON und vergleichen Sie die disassemblierten Befehle mit den ursprünglichen Werten, die Sie in Listing 24 finden. Sie werden erkennen, daß die Befehle »6002 LDA 2000,Y« und »6007 STA 2000,Y« aufgrund der INC-Befehle immer auf andere Adressen zugreifen. Besagte INC-Befehle erhöhen jeweils das High-Byte des Operanden. Ist dieses schon $40, so wird die Schleife beendet. In Listing 25 sehen Sie, wie unsere Schleife aus Listing 24 aussieht, wenn sie fertig durchlaufen wurde. Ein weiterer Start bewirkt, daß das Pro-

```
100 -.BA $0801
110 -.OB "LOADER-MAKER 64,P,W"
120 -;
130 -;
140 -; ****************************
150 -; *                          *
160 -; *  L O A D E R - M A K E R *
170 -; *                          *
180 -; ****************************
190 -; *                          *
200 -; *   EIN PROGRAMMGENERATOR  *
210 -; *                          *
220 -; *    VON FLORIAN MUELLER   *
230 -; *                          *
240 -; ****************************
250 -;
260 -;
270 -;
280 -.GL BASIN  = $FFCF
290 -.GL SETPAR = $FFBA
300 -.GL SETNAM = $FFBD
310 -.GL LOAD   = $FFD5
320 -.GL READY  = $A474
330 -.GL NUMOUT = $BDCD
340 -.GL TASTPF = 631 ; TASTATURPUFFER
350 -.GL ANZAHL = 198 ; ENTHAELT ANZAHL
360 -;                  DER ZEICHEN IM
370 -;                  TASTATURPUFFER
380 -.GL KASSPF = 828 ; KASSETTENPUFFER
390 -;
400 -;
410 -.MA PRINT (TEXT)
420 -          LDA #<(TEXT)    ; MAKRO
430 -          LDY #>(TEXT)    ; FUER
440 -          JSR $AB1E       ; TEXTAUSGABE
450 -.RT
460 -;
470 -;
480 -;
490 -;
500 -.WO LINK+1 ; LINKPOINTER
510 -.WO 1985   ; ZEILENNUMMER
520 -.BY $9E    ; TOKEN FUER "SYS"
530 -          .TX "2061"
540 -LINK     .BY 0,0,0       ; ENDMARKIERUNG
550 -;                         DER BASIC-ZEILE
560 -;
570 -SYSTEM   LDX #0          ; FLAG FUER SYSTEM-
580 -         STX $9D         ; MELDUNGEN SETZEN
590 -;
600 -         LDX #$49        ; DEKR.-ZAEHLER
610 -SCHLEIFE1 LDA ABLAGE,X   ; LADEROUTINE
620 -         STA KASSPF,X    ; VON ABLAGE IN
630 -         DEX             ; DEN BEREICH
640 -         BPL SCHLEIFE1   ; KOPIEREN, IN
650 -;                         DEM SIE LAEUFT
660 -         JMP KASSPF      ; & STARTEN
670 -;
680 -;
690 -; ES FOLGT DIE LADEROUTINE, DIE HIER
700 -; AN FALSCHER STELLE ABGELEGT IST UND
710 -; VON DER "SCHLEIFE1" (600-640) IN
720 -; DEN ORIGINALBEREICH GESCHRIEBEN WIRD.
730 -;
740 -ABLAGE   LDA #1          ; FILENUMMER #1
750 -         TAY             ; SEKUNDAERADRESSE #1
760 -GERAETENR LDX #0         ; GERAETEADRESSE #?
770 -         JSR SETPAR      ; PARAMETER SETZEN
780 -;
790 -LAENGE   LDA #0          ; LAENGE DES FILENAMEN
800 -         LDX #<($35C)    ; ADRESSE DES
810 -         LDY #>($35C)    ; FILENAMEN: $035C
820 -         JSR SETNAM      ; NAMEN SETZEN
830 -;
840 -         LDA #0          ; FLAG FUER "LADEN"
850 -         JSR LOAD
860 -;
870 -FEHLER   BCS LOADERROR   ; LADEFEHLER?
880 -START    JMP 0           ; ZUR STARTADRESSE
890 -LOADERROR LDX #$1D       ; "LOAD ERROR"
900 -         JMP ($300)      ; AUSGEBEN
910 -;
920 -NAME     .BY 0,0,0,0     ; 16 BYTES
930 -         .BY 0,0,0,0     ; FUER FILENAMEN
940 -         .BY 0,0,0,0     ; RESERVIEREN
950 -         .BY 0,0,0,0
960 -;
970 -BASIC    STX $2D         ; POINTER FUER
980 -         STY $2E         ; PROGRAMMENDE SETZEN
990 -         JSR $E544       ; = PRINT CHR$(147)
1000 -        LDX #3          ; 3 BYTES IN
1010 -        STX ANZAHL      ; TASTATURPUFFER
1020 -;
1030 -SCHLEIFE2 LDA $0383,X    ; AUS DER TABELLE
1040 -        STA TASTPF,X    ; IN ZEILE 1100
1050 -        DEX             ; KOPIEREN
1060 -        BPL SCHLEIFE2
1070 -;
1080 -        JMP READY       ; WARMSTART
1090 -;
1100 -.BY "R",$D5,13 ; "R",SHIFT U,RETURN
1110 -;
1120 -; HIER ENDET DER PROGRAMMTEIL,
1130 -; DER MODIFIZIERT WIRD.
1140 -; ES FOLGT DIE MODIFIKATIONSROUTINE:
1150 -;
1160 -MDFIKATOR JSR $E544      ; = PRINT CHR$(147)
1170 -...PRINT (TEXT1)
1180 -; STARTADRESSE HOLEN
1190 -;
1200 -        JSR $AEFD       ; PRUEFT AUF KOMMA
1210 -        JSR $AD8A       ; HOLT PARAMETER
1220 -        JSR $B7F7       ; NACH $14/$15
1230 -;
1240 -        LDX $14         ; STARTADRESSE
1250 -        LDA $15         ; HOLEN,
1260 -        STX START+1     ; IM PROGRAMM
1270 -        STA START+2     ; ABLEGEN UND
1280 -        JSR NUMOUT      ; UND AUSGEBEN
1290 -;
1300 -;
1310 -; NUN WIRD NOCH DER ZU MODIFIZIERENDE
1320 -; PROGRAMMTEIL IN DEN AUSGANGSZUSTAND
1330 -; GEBRACHT:
1340 -;
1350 -        LDX #15         ; NAMEN MIT NULL-BYTES
1360 -        LDA #0          ; BELEGEN
1370 -SCHLEIFE3 STA NAME,X     ; DURCH EINE
1380 -        DEX             ; DEKREMENTIER-
1390 -        BPL SCHLEIFE3   ; SCHLEIFE
1400 -;
1410 -        STA SYSTEM+1    ; KEINE SYSTEMMELDUNGEN
1420 -;
1430 -        LDA #3          ; SPRUNGWEITE = 3
1440 -        STA FEHLER+1
1450 -;
1460 -        LDA #$A2        ; OPCODE FUER "LDX #"
1470 -        STA GERAETENR
1480 -;
1490 -;
1500 -; AN DIESER STELLE IST DAS "GERUEST"
1510 -; (DER ZU MODIFIZIERENDE TEIL)
1520 -; IM AUSGANGSZUSTAND
1530 -;
1540 -;
1550 -; EINGABE DES FILENAMEN
1560 -; ====================
1570 -;
1580 -...PRINT (TEXT2)
1590 -        LDX #0          ; ZAEHLER AUF 0
1600 -SCHLEIFE4 JSR BASIN
1610 -        CMP #13         ; ENDE DER EINGABE?
1620 -        BEQ WEITER1     ; JA=>WEITER
1630 -        STA NAME,X      ; BYTE ABLEGEN
1640 -        INX
1650 -        CPX #16         ; 16 ZEICHEN MAX.
1660 -        BNE SCHLEIFE4   ; NAECHSTES ZEICHEN
1670 -;
1680 -; WENN DIESE STELLE DURCHLAUFEN WIRD,
1690 -; HAT DAS X-REGISTER DEN WERT 16.
1700 -;
1710 -; BEI "WEITER1" HINGEGEN KANN ES AUFGRUND
1720 -; DES BRANCH-BEFEHLS "BEQ WEITER1"
1730 -; UNTERSCHIEDLICHE WERTE HABEN.
1740 -;
1750 -WEITER1  STX LAENGE+1
1760 -;
1770 -;
1780 -; EINGABE DER GERAETEADRESSE
1790 -; =========================
1800 -;
1810 -...PRINT (TEXT3)
1820 -        JSR BASIN       ; HOLT ZEICHEN
1830 -        SEC             ; VOR SUBTRAKTION
1840 -        SBC #"0"        ; IM AKKU STEHT JETZT
1850 -;                         DIE ZAHL
1860 -;
1870 -        STA GERAETENR+1; ABLEGEN
1880 -        BNE WEITER2     ; GERAET<>0 : WEITER
1890 -; DA ALS GERAETENUMMER 0 EINGEGEBEN
1900 -; WURDE, MUSS DER GESAMTE BEFEHL
1910 -; "LDX #GERAET" IN "LDX $BA"
1920 -; UMGEWAENDELT WERDEN, DAMIT DAS
1930 -; NACHLADEN VON DEM GERAET ERFOLGT,
1940 -; VON DEM DER LADER EINGELESEN WIRD.
1950 -;
1960 -        LDA #$A6        ; OPCODE FUER "LDX ZP"
1970 -        STA GERAETENR
1980 -        LDA #$BA        ; "LDX $BA"
1990 -        STA GERAETENR+1; GENERIEREN
2000 -;
2010 -;
2020 -; MASCHINENPROGRAMM (J/N)?
2030 -; =======================
```

Listing 30

```
2040 -;
2050 -WEITER2   ... PRINT(TEXT4)
2060 -          JSR JANEIN     ; (JA/NEIN)?
2070 -          BEQ WEITER3    ; JA=>WEITER
2080 -          LDA #$6C       ; SPRUNG AUF $036C
2090 -          LDY #$03       ; VERBIEGEN
2100 -          STA START+1    ; BEI $36C STEHT
2110 -          STY START+2    ; EINE ROUTINE,
2120 -;                          DIE DEN "RUN"-
2130 -;                          BEFEHL SIMULIERT
2140 -;
2150 -;
2160 -; SYSTEMMELDUNGEN (J/N)?
2170 -; ======================
2180 -;
2190 -WEITER3   ... PRINT(TEXT5)
2200 -          JSR JANEIN     ; (JA/NEIN)?
2210 -          BNE WEITER4    ; NEIN=>WEITER
2220 -          LDA #$80       ; FLAG FUER
2230 -          STA SYSTEM+1   ; SYSTEMMELDUNGEN
2240 -;
2250 -;
2260 -; LOAD ERROR  AUSGEBEN (J/N)
2270 -; ==========================
2280 -;
2290 -;
2300 -WEITER4   ... PRINT(TEXT6)
2310 -          JSR JANEIN     ; (JA/NEIN)?
2320 -          BEQ WEITER5    ; NEIN=>WEITER
2330 -          LDA #0         ; FEHLERMELDUNGEN
2340 -          STA FEHLER+1   ; UNTERDRUECKEN
2350 -;
2360 -;
2370 -; PROGRAMMENDE
2380 -; ============
2390 -;
2400 -;
2410 -WEITER5   ... PRINT(TEXT7)
2420 -;
2430 -; VEKTOR FUER BASIC-ENDE SETZEN
2440 -; =============================
2450 -;
2460 -;
2470 -          LDA #<(MDFIKATOR)
2480 -          STA $2D        ; LOW-BYTE
2490 -          LDA #>(MDFIKATOR)
2500 -          STA $2E        ; HIGH-BYTE
2510 -          JMP READY      ; SPRUNG INS BASIC
2520 -;
2530 -;
10000-;
10010-; ASCII-TABELLEN
10020-; ==============
10030-;
10040-;
10050-TEXT1     .TX "LOADER-MAKER 64"
10060-          .BY 13,13
10070-          .TX "STARTADRESSE : "
10080-          .BY 0
10090-;
10100-TEXT2     .BY 13,13
10110-          .TX "FILENAME : "
10120-          .BY 0
10130-;
10140-TEXT3     .BY 13,13
10150-          .TX "GERAETENR. (1-9;0=UEBERNEHMEN) : "
10160-          .BY 0
10170-;
10180-TEXT4     .BY 13,13
10190-          .TX "MASCHINENPROGRAMM"
10200-          .BY 0
10210-;
10220-TEXT5     .BY 13,13
10230-          .TX "SYSTEMMELDUNGEN"
10240-          .BY 0
10250-;
10260-TEXT6     .BY 13,13
10270-          .TX "LOAD ERROR  AUSGEBEN"
10280-          .BY 0
10290-;
10300-TEXT7     .BY 13,13,18
10310-          .TX "*** LOADER GENERIERT ***"
10320-          .BY 13,13
10330-          .TX "MIT 'SAVE' SPEICHERN,"
10340-          .TX " MIT 'RUN' STARTEN"
10350-          .BY 0
10360-;
10370-TEXT8     .BY 13,13,18
10380-          .TX "*** PROGRAMMENDE ! ***"
10390-          .BY 13,13,0
10400-;
10410-TEXT9     .TX " (J/N)? "
10420-          .BY 0
10430-;
10440-;
20000-;
20010-; UNTERPROGRAMM FUER "J/N?"
20020-; ========================
20030-;
20040-;
20050-JANEIN    ... PRINT(TEXT9)
20060-          JSR BASIN      ; EINGABE HOLEN
20070-          CMP #"←"
20080-          BNE JANEIN1
20090-          PLA            ; SIEHE STAPEL-
20100-          PLA            ; MANIPULATION
20110-...PRINT(TEXT8)
20120-          JMP READY      ; SPRUNG INS BASIC
20130-JANEIN1   CMP #"J"       ; VERGLEICH MIT "J"
20140-          RTS            ; RUECKKEHR VOM
20150-;                          UNTERPROGRAMM
20160-.EN
```

Listing 30 (Schluß)

gramm sich früher oder später selbst invertiert und darum abstürzt.

Was nämlich unserem Listing 24 fehlt, damit es mehr als einmal arbeitet, ist eine Initialisierung, die jedesmal den Ausgangswert ($2000) in die LDA/STA-Befehle einsetzt. In Listing 26 sehen Sie eine solche Initialisierung (6000 - 600F). Die Adresse $FFFF (bei 6012 und 6017) ist ein Dummy-Wert, das heißt er dient nur zum vorläufigen Ausfüllen von Adressen und hat keine programmtechnische Bedeutung. Der Dummy-Wert wird ohnehin von der Initialisierung überschrieben; wir hätten also statt $FFFF auch $040C oder andere verwenden können. Wichtig ist nur, daß »LDA Dummy,Y« 3 Byte belegt.

Ein besonderer Vorteil der Selbstmodifikation ist es, daß selbstmodifizierende Schleifen keine Zähler in der Zeropage benötigen, weil der Zähler praktisch im Programm selbst liegt. In puncto Geschwindigkeit sind selbstmodifizierende Schleifen den herkömmlichen aber oft unterlegen.

Ein weiterer Vorteil von ihnen ist aber, daß man außer mit weniger Zeropage-Speicherplätzen auch mit weniger Registern auskommen kann (sofern man hier Einsparungen vornehmen will). Listing 27 beispielsweise invertiert den Bereich $3FD2 - $475F. X- und Y-Register sowie die Zeropage bleiben unverändert, lediglich der Akkumulator fungiert als Arbeitsregister.

Listing 28 kopiert den Basic-Interpreter ($A000 - $BFFF) ins RAM an gleicher Adresse, wobei nur das X-Register verwendet wird (!).

Nun wollen wir sehen, wie man bei der Entwicklung selbstmodifizierender Programme unter Zuhilfenahme eines guten Assemblers (Hypra-Ass) vorgehen muß.

Zunächst einmal müssen diejenigen Stellen, an denen Modifikationen vorgenommen werden, mit Labels definiert werden. Von diesen Labels aus können die Stellen im Speicher, die geändert werden sollen, leicht berechnet werden.

Befehlscode	= LABEL + 0 =	LABEL
Low-Operand	=	LABEL + 1
High-Operand	=	LABEL + 2

Bei 2-Byte-Befehlen wird der Parameter wie der Low-Operand eines 3-Byte-Befehls errechnet.

Als Beispiel finden Sie in Form von Listing 29 einen Quelltext (Assembler: Hypra-Ass) für Listing 28. Während in Listing 28 der Ausgangswert bei 6010 »LDX 0000« und bei 6013 »STX 0000« ist, wurde im Quelltext $FFFF verwendet (270, 280), um den Assembler zu zwingen, den Dummy-Wert als 16-Bit-Adresse abzulegen (und nicht als Zeropage-Adresse, wodurch der Befehl nur 2 statt 3 Byte belegen würde).

Die Stellen, die modifiziert werden, wurden mit »MOD1« und »MOD2« definiert. MOD1 ist zugleich der Schleifenbeginn.

Nachdem Sie jetzt den Eingang gefunden haben, möchte ich einige Anregungen liefern, wie Sie die Vorteile der Selbstmodifikation nutzen können. Wir werden hier die Anwendung nach den verschiedenen Adressierungsarten unterteilen.

a) Anwendung auf absolute Adressierung

Bei der Stapelmanipulation haben wir schon ein Verfahren kennengelernt, den Befehl JSR (indirekt), der im normalen 6510-Befehlssatz nicht existiert, zu simulieren.

Folgendermaßen kann über Selbstmodifikation ein Unterprogramm ab ADRESSE aufgerufen werden.

```
                LDA #<ADRESSE
                STA SPRUNGBEFEHL+1 ; Low-Operand
                LDA #>ADRESSE
                STA SPRUNGBEFEHL+2; High-Operand
SPRUNGBEFEHL JSR $FFFF ; $FFFF=Dummy
```

Genauso kann man mit dem JMP-Befehl verfahren. Sogar bei den Schieber-, Dekrementier- und Inkrementierbefehlen, die im Gegensatz zu JMP die indirekte Adressierung nicht haben, ist auf diese Weise eine Simulation der indirekten Adressierung möglich.

Wird eine Sprungtabelle per Selbstmodifikation verarbeitet, müssen die Sprungadressen in der Tabelle nicht (!) dekrementiert werden.

b) Anwendung auf Immediate-Befehle

Oft müssen berechnete Werte auf dem Stapel oder im Speicher abgelegt und im Gebrauchsfall wieder aufgenommen werden.

Ein Beispiel hierfür ist der »Basic-Start-Generator« (64'er, 7/85, Seite 74). Bei Erwähnung dieses Programms taucht natürlich die Frage auf, ob es sich hier noch um ein selbstmodifizierendes Programm handelt oder ob der »Basic-Start-Generator« nicht eher zu den Programmgeneratoren zählt. Wir wollen darauf kurz eingehen.

Der »Basic-Start-Generator« ist eindeutig den Programmgeneratoren zuzuordnen, da der generierte Programmteil nie angesprungen wird und somit ein eigenständiges Programm darstellt. Das Programm modifiziert also nicht sich selbst, sondern vielmehr ein zweites Programm, welches dann vom Benutzer gespeichert werden kann.

Die Programmierung ist aber bei Programmgeneratoren nicht anders als bei selbstmodifizierenden Programmen. Auf den Unterschied Programmgeneration/Selbstmodifikation werden wir an späterer Stelle näher eingehen.

Zunächst wollen wir aber ein praktisches Beispiel für die Anwendung der Modifikation von Immediate-Befehlen behandeln. Oft steht man vor dem Problem, ein Register zu sichern und später wieder zu holen. Im Falle des Akkumulators sieht das so aus:

```
        PHA          ; Akku sichern
        .....        ; weiteres Programm
        PLA          ; Akku wieder holen
```

Beim X-Register wird's schon ungünstiger:

```
        TXA          ; X-Register in Akku
        PHA          ; Akku sichern
        ......       ; weiteres Programm
        PLA          ; Akku wieder holen
        TAX          ; Akku ins X-Register
```

Hier wird also zusätzlich der Akku beeinflußt. Wenn dies vermieden werden muß, wird folgender Weg gewählt:

```
        STX $02      ; $02 = Zwischenspeicher
        ....         ; weiteres Programm
        LDX $02      ; X wieder holen
```

Für die Sicherung des X-Registers gibt es aber noch eine weitere Lösung, die den X-Wert im Programm ablegt und dadurch nicht den Stapel oder einen Zwischenspeicher außerhalb des Programms benötigt.

```
        STX GETX+1 ; X direkt in Immediate-Befehl
                     schreiben
        ....         ; weiteres Programm
GETX    LDX #$00     ; $00 = Dummy-Wert
```

Obiges Beispiel kann sehr leicht auf Akkumulator oder Y-Register umgeschrieben werden.

Folgendermaßen kann das X-Register mit dem Akkumulator verglichen werden:

```
        STX VGL+1    ; in Vergleichsbefehl ablegen
        (........    ; evtl. weitere Programme)
VGL     CMP #$00     ; $00 = Dummy
```

```
programm : loader-maker      0801 0a38
--------------------------------------

0801 : 0b 08 c1 07 9e 32 30 36  0a
0809 : 31 00 00 00 a2 00 86 9d  ba
0811 : a2 49 bd 1f 08 9d 3c 03  0f
0819 : ca 10 f7 4c 3c 03 a9 01  f7
0821 : a8 a2 00 20 ba ff a9 00  71
0829 : a2 5c a0 03 20 bd ff a9  c5
0831 : 00 20 d5 ff b0 03 4c 00  0b
0839 : 00 a2 1d 6c 00 03 00 00  77
0841 : 00 00 00 00 00 00 00 00  42
0849 : 00 00 00 00 00 00 86 2d  be
0851 : 84 2e 20 44 e5 a2 03 86  09
0859 : c6 bd 83 03 9d 77 02 ca  72
0861 : 10 f7 4c 74 a4 52 d5 0d  5d
0869 : 20 44 e5 a9 21 a0 09 20  d5
0871 : 1e ab 20 fd ae 20 8a ad  9e
0879 : 20 f7 b7 a6 14 a5 15 8e  37
0881 : 38 08 8d 39 08 20 cd bd  7c
0889 : a2 0f a9 00 9d 3f 08 ca  a7
0891 : 10 fa 8d 0e 08 a9 03 8d  38
0899 : 36 08 a9 a2 8d 22 08 a9  ef
08a1 : 42 a0 09 20 1e ab a2 00  44
08a9 : 20 cf ff c9 0d f0 08 9d  9e
08b1 : 3f 08 e8 e0 10 d0 f1 8e  b7
08b9 : 28 08 a9 50 a0 09 20 1e  69
08c1 : ab 20 cf ff 38 e9 30 8d  1f
08c9 : 23 08 d0 0a a9 a6 8d 22  b0
08d1 : 08 a9 ba 8d 23 08 a9 74  10
08d9 : a0 09 20 1e ab 20 1b 0a  06
08e1 : f0 0a a9 6c a0 03 8d 38  97
08e9 : 08 8c 39 08 a9 88 a0 09  fa
08f1 : 20 1e ab 20 1b 0a d0 05  5f
08f9 : a9 80 8d 0e 08 a9 9a a0  81
0901 : 09 20 1e ab 20 1b 0a f0  fc
0909 : 05 a9 00 8d 36 08 a9 b1  42
0911 : a0 09 20 1e ab a9 69 85  ba
0919 : 2d a9 08 85 2e 4c 74 a4  2e
0921 : 4c 4f 41 44 45 52 2d 4d  24
0929 : 41 4b 45 52 20 36 34 0d  4a
0931 : 0d 53 54 41 52 54 41 44  7a
0939 : 52 45 53 53 45 20 3a 20  ec
0941 : 00 0d 0d 46 49 4c 45 4e  7d
0949 : 41 4d 45 20 3a 20 00 0d  45
0951 : 0d 47 45 52 41 45 54 45  b8
0959 : 4e 52 2e 20 28 31 2d 39  93
0961 : 3b 30 3d 55 45 42 45 52  ce
0969 : 4e 45 48 4d 45 4e 29 20  c1
0971 : 3a 20 00 0d 0d 4d 41 53  44
0979 : 43 48 49 4e 45 4e 50 52  a9
0981 : 4f 47 52 41 4d 4d 00 0d  8a
0989 : 0d 53 59 53 54 45 4d 4d  40
0991 : 45 4c 44 55 4e 47 45 4e  89
0999 : 00 0d 0d 4c 4f 41 44 20  3d
09a1 : 45 52 52 4f 52 20 20 41  b7
09a9 : 55 53 47 45 42 45 4e 00  aa
09b1 : 0d 0d 12 2a 2a 2a 20 4c  1c
09b9 : 4f 41 44 45 52 20 47 45  30
09c1 : 4e 45 52 49 45 52 54 20  e8
09c9 : 2a 2a 2a 0d 0d 4d 49 54  3e
09d1 : 20 27 53 41 56 45 27 20  ee
09d9 : 53 50 45 49 43 48 45 52  ff
09e1 : 4e 2c 20 4d 49 54 20 27  fd
09e9 : 52 55 4e 27 20 53 54 41  cf
09f1 : 52 54 45 4e 00 0d 0d 12  49
09f9 : 2a 2a 2a 20 50 52 4f 47  2a
0a01 : 52 41 4d 4d 45 4e 44 45  53
0a09 : 20 21 20 2a 2a 2a 0d 0d  49
0a11 : 00 20 28 4a 2f 4e 29 3f  fd
0a19 : 20 00 a9 12 a0 0a 20 1e  fd
0a21 : ab 20 cf ff c9 5f d0 0c  c3
0a29 : 68 68 a9 f6 a0 09 20 1e  1e
0a31 : ab 4c 74 a4 c9 4a 60 5c  dd
```

Listing 31

Als letztes Beispiel soll das Y-Register zum Akkumulator addiert werden:

```
    STY ADD+1  ; in Arithmetikbefehl ablegen
    (.........  ; evtl. weiteres Programm)
    CLC        ; Carry vor Addition
ADD ADC #$FF   ; $FF = Dummy
```

Die Anwendungsmöglichkeiten sind hier unbegrenzt.

c) Anwendung auf komplette Befehle

Bisher haben wir nur die Parameter einzelner Befehle modifiziert. Es ist selbstverständlich auch möglich, die Befehlscodes oder die kompletten Befehle zu modifizieren.

Wenn nur der Befehlscode geändert wird (zum Beispiel ein ORA #- in einen EOR #-Befehl) bleiben die Parameter erhalten. Es könnte ferner ein impliziter Befehl (SEI,CLI, CLD, DEX,INX,....) geändert werden, um beispielsweise zwischen In- und Dekrementieren umzuschalten. Außerdem könnte bei einem BRANCH-Befehl die Sprungbedingung (CS,CC,VS, VC,NE,EQ) geändert werden. Aus BCS könnte also leicht BCC werden.

Weil man hier die Opcodes der Befehle kennen muß, empfehle ich dazu die Tabelle 24 in »Assembler ist keine Alchimie«.

Nun lösen wir noch das häufig auftretende Problem, wie die Ausführung eines Unterprogramms verhindert wird. Dazu werden wir drei Lösungen (I - III) entwickeln.

I. Die Adresse FLAG wird auf 0 gesetzt, wenn das Unterprogramm ausgeführt werden soll; auf einen anderen Wert, wenn es nicht ausgeführt werden soll.

```
     LDA #0        ; Flag für Ausführung
     STA FLAG      ; Flag setzen
     (........     ; evtl. weiteres Programm)
     LDA FLAG      ; Flag testen
     BNE NEIN      ; Flag < > 0, also nicht ausführen
     JSR UNTER-
     PROGRAMM      ; Aufruf
NEIN ........        weiteres Programm
```

Das Flag könnte auch am Beginn des Unterprogramms abgefragt und dann (wenn FLAG < > 0) das Unterprogramm verlassen werden.

II. Als ersten Befehl des Unterprogramms verwenden wir NOP:

```
UP   NOP     ; Beginn des Unterprogramms
     ......  ; Fortsetzung des Unterpro-
               gramms
```

So wird die Ausführung des Unterprogramms gestattet:

```
     LDA #$EA  ; Opcode für NOP
     STA UP    ; an Anfang des Unterprogramms
                 schreiben
```

Und so wird sie verhindert:

```
     LDA #$60  ; Opcode für RTS
     STA UP    ; an Anfang des Unterprogramms
                 schreiben
```

Wer noch einen NOP-Befehl und damit 1 Byte sparen möchte, kann den NOP-Befehl entfallen lassen. Dann muß auch der Opcode $EA beim Erlauben des Unterprogramms in den Opcode des ersten Byte im Unterprogramm geändert werden. Weil dies ziemlich mühselig ist, ziehe ich die ursprüngliche Lösung II trotz des um 1 Byte erhöhten Speicherbedarfs vor.

III. Das beste Verfahren. Wir schalten den JSR-Befehl aus, indem wir ihn in einen BIT-Befehl abändern.

```
AUFRUF  JSR Unterprogramm
```

JSR ausschalten:

```
        LDA #$2C     ; Opcode für BIT
        STA AUFRUF
```

JSR wieder erlauben:

```
        LDA #$20     ; Opcode für JSR
        STA AUFRUF
```

Der JSR-Opcode kann auch mit $0C überschrieben werden. $0C ist ein illegaler Opcode für ein 3-Byte-NOP und arbeitet mit allen mir bekannten Versionen des C 64. Ob er ebenfalls auf dem C 128 läuft, konnte ich noch nicht prüfen.

Im übrigen können mit dem soeben beschriebenen Verfahren auch andere Befehle ausgeschaltet werden, zum Beispiel JMP, LDA, STA und so weiter. Wenn aber der JSR-Opcode mit $2C (BIT) überschrieben wird, ist darauf zu achten, daß bei der Ausführung des BIT-Befehls die Prozessorflags gesetzt werden.

Sicherlich gibt es noch mehr Problemlösungen als I - III, aber III dürfte wohl kaum zu übertreffen sein.

d) Anwendung auf mehrere Befehle

Selbstverständlich können ganze Befehlsfolgen, also größere Programmteile gegeneinander ausgetauscht werden. Zu beachten ist nur, daß die Routinen, die gegeneinander ausgetauscht werden, auch in dem Bereich, in den sie vom Programm aus geschrieben werden, lauffähig sind. Dies ist vor allem dann gegeben, wenn nur die relative Adressierung verwendet wird und dadurch die Routine im Speicher frei verschoben werden kann.

e) Anwendung auf Tabellen

Dieser Anwendungsfall würde auch zum Abschnitt über »Tabellen« passen.

Wir bleiben hier bei der Theorie, denn die Umsetzung in ein Programm ist nicht mehr schwer. Vielmehr soll Ihre Kreativität nicht durch Unmengen von Beispielen gehemmt werden.

Zunächst wollen wir uns ein wenig mit dem SMON befassen. Wenn Sie den Disk-Monitor einschalten, kopiert das Programm einen Floppy-Befehl («U1 ..«) vom Ende des SMON in einen Bereich zwischen $02A0 und $02FF. Dieser Lesebefehl wird nach Bedarf modifiziert, zum Beispiel wird beim Schreiben der »U1«- in einen »U2«-Befehl umgewandelt oder die Angabe des einzulesenden Blocks wird geändert. Dies wäre ein typisches Anwendungsbeispiel für Selbstmodifikation, wenn der Lesebefehl nicht erst in einen Bereich außerhalb des Programms kopiert würde (worin ich keinen Sinn sehe), sondern am Ende des SMON (etwa bei $CFF0) bliebe und dort modifiziert würde.

Im Hi-Eddi liegt eine Tabelle, die die High-Byte der Bit-Map-Anfangsadressen beinhaltet. Diese Tabelle wird von Hi-Eddi bei jedem Bildwechsel umgerechnet.

Nach den vorausgegangenen zwei Beispielen an Spitzenprogrammen aus dem 64'er möchte ich noch andere Anwendungsbeispiele nennen.

Besonders flexible Programme erlauben Eingriffe des Anwenders in die Befehls- oder Text-Tabellen. So können Bildschirmmasken editiert oder Eingabemasken erstellt werden.

Ein solches Programm braucht sich nach den Modifikationen nur selbst abzuspeichern. Weil hier unter Umständen ein erheblicher Teil des Programmschutzes verlorengeht, werden dann lediglich die Tabellen gespeichert.

Ein Adventure-Generator modifiziert in der Regel auch nur die Tabellen eines fertigen Adventure-Programms, das eigentliche Programm bleibt unverändert. In diesen Tabellen sind die einzelnen Spielsituationen enthalten.

Bei diesen (theoretischen) Fällen wollen wir es belassen. Letztendlich muß ja der Programmierer entscheiden, inwieweit er die Selbstmodifikation auf Tabellen anwenden kann.

f) Das Beispielprogramm »Loader-Maker 64«

Wie aus dem Namen des Beispielprogramms schon zu entnehmen ist, handelt es sich um einen Programmgenerator. Da - wie gesagt - die Programmierung wie bei selbstmodifizierenden Programmen ist, habe ich bewußt einen Programmgenerator als Beispiel gewählt.

Als Listing 31 finden Sie ein MSE-Listing, falls Sie »Loader-Maker 64« bequem abtippen wollen und an der Anwendung des Programms interessiert sind. Deshalb zunächst eine Kurzbeschreibung für Anwender:

»Loader-Maker« ermöglicht es Ihnen, zu einem Programm ein (Maschinensprache-) Ladeprogramm zu generieren, welches normal geladen und mit »RUN« gestartet wird, worauf es das nachzuladende Programm nachlädt und startet.

Nach dem Laden von »Loader-Maker« wird dieses Programm durch »SYS 2154,START« gestartet. START ist eine Variable und wird durch die Startadresse des nachzuladenden Programms ausgedrückt. Soll ein Basic-Programm nachgeladen werden, hat diese Adresse keine Bedeutung (einfach SYS2154,0 eingeben). Bei einem Maschinenprogramm handelt es sich hier um die Adresse, mit der das Programm über »SYS« gestartet wird (49152 beim SMON $C000).

Das Programm meldet sich mit »Loader-Maker 64« und gibt die Startadresse aus. Dazu können Sie den Filenamen eingeben.

Bei allen weiteren Eingaben (Gerätenummer, von der geladen werden soll; Maschinenprogramm j/n; Systemmeldungen wie »SEARCHING FOR« ausgeben j/n; »LOAD ERROR« bei Ladefehler ausgeben j/n) können Sie das Programm durch Eingabe des Linkspfeils abbrechen. Sind alle Eingaben erledigt, kommt die Meldung »LOADER GENERIERT« und der Lader kann mit »SAVE« gespeichert werden.

Wenn das nachzuladende Programm von der Adresse geladen werden soll, von der auch das Ladeprogramm selbst eingelesen wurde, ist als Gerätenummer nur 0 einzugeben.

Befassen wir uns nun mit dem Programm, dessen Quelltext Sie als Listing 30 finden.

Die Zeilen bis 990 stellen das Ladeprogramm in unmodifizierter Form dar und enthalten viele Dummywerte, wie zum Beispiel die (unsinnige) Startadresse 0 in Zeile 880.

Mit 1160 beginnt die Modifikationsroutine. Nach 1220 wurde die Startadresse eingelesen, die ja per SYS übergeben wurde, und wird wieder mit dem Titel ausgegeben. 1260/1270 schreiben die Startadresse hinter den JMP-Befehl in Zeile 880.

1350 - 1470 bringen das (noch unmodifizierte) Gerüst in den Ausgangszustand, der dann nach Bedarf geändert wird.

1580 - 1660 holen den Filenamen, legen ihn bei NAME (920) ab, berechnen gleich die Länge des Filenamens und legen diese bei LAENGE (790) ab.

1810 - 1990 holen die Geräteadresse. Da diese im ASCII-Format vorliegt, muß der ASCII-Code von 0 abgezogen werden (1830/1840). Wurde 0 eingegeben, wird der LDX #DEVICE-Befehl (760) in »LDX $BA« geändert. Die Adresse $BA enthält jeweils die Adresse, von der das letzte Programm geladen wurde.

2050 - 2110 fragen, ob das nachzuladende Programm mit der per SYS übermittelten Startadresse gestartet wird (Eingabe »j«). Wurde »n« eingegeben, muß das Programm über den Basic-Befehl »RUN« eingegeben werden. Auf eine entsprechende Routine wird die Startadresse gestellt (2080 - 2110).

2190 - 2230 ermöglichen die Einstellung, ob »SEARCHING..«, »LOADING« etc. ausgegeben werden sollen.

Soll im Falle eines Ladefehlers das Programm nicht gestartet und stattdessen »LOAD ERROR..« ausgegeben werden, wird dies bei 2300 - 2340 festgelegt. Wird die Fehlerausgabe unterdrückt, muß der BCS-Befehl (870) unschädlich gemacht werden. Dies geschieht einfach dadurch, daß die Sprungweite auf 0 gesetzt wird (2330/2340).

Am Programmende wird noch eine Meldung ausgegeben (2410 - 2510) und der Vektor für das Ende des Basic-Programms neu gesetzt, damit das generierte Ladeprogramm mit »SAVE« gespeichert werden kann.

10000 - 10420 enthalten nur die Text-Tabellen.

Von 20050 bis zur letzten Zeile (20140) steht ein Unterprogramm, das bei jeder J/N-Entscheidung über »JSR J,N« aufgerufen wird.

Es gibt den Text »(J/N)?« aus (20050) und holt eine Eingabe. Ist diese »J«, so ist nach dem Verlassen des Unterprogramms (20140) das Zero-Flag gesetzt (andernfalls nicht).

Wurde der Linkspfeil eingegeben, wird das Programm abgebrochen und eine entsprechende Meldung ausgegeben (20070 - 20120).

Wie wir nun gesehen haben, handelt es sich bei »Loader-Maker« um einen Programmgenerator. Mit zwei kleinen Änderungen wird er jedoch zum selbstmodifizierenden Ladeprogramm. Wir müssen nur die beiden »JMP READY.«-Befehle (2510/20120) in »JMP SYSTEM« umwandeln, wodurch am Programmende der generierte Lader angesprungen würde. Schon hätten wir ein selbstmodifizierendes Ladeprogramm.

Um Ihnen noch die Anwendung des Loader-Maker zu erleichtern, hier zwei Eingabebeispiele:

```
Startadresse ..................49152
Filename.......................SMON $C000
Geräteadresse ................0
Maschinenprogramm ......j
Systemmeldungen ..........j
LOAD ERROR ausgeben j

Startadresse ..................0 (bedeutungslos)
Filename.......................HI-EDDI
Geräteadresse ................8
Maschinenprogramm ......n
Systemmeldungen ..........n
LOAD ERROR ausgeben j
```

g) Verbesserungen an »Tabellen-Beispiel«

Zum Abschluß des Themas »Selbstmodifikation« wollen wir noch kleine Verbesserungen am Programm »Tabellen-Beispiel« erwähnen. Ich werde Ihnen eher Anregungen geben als fertige Änderungsvorschläge.

Zunächst soll die Adresse XSAVE (zum Sichern des X-Registers in Schleifen) überflüssig werden. So könnte es nun gesichert werden:

```
XSV   STX  GETX+1
.......
GETX  LDX  #$00       0=Dummy; hier wird X wieder
                        aufgenommen.
```

Auch die Sprungtabelle läßt sich - viel einfacher, finde ich - anders handhaben:

```
      LDA  J?LO,X     JMLO oder JELO
      STA  SPRG+1
      LDA  J?HI,X     JMHI oder JEHI
      STA  SPRG+2
SPRG  JMP  0000
```

In den Tabellen JMLO/JMHI und JELO/JEHI (Low- und High-Bytes der Sprungadressen) dürfen die Adressen aber nicht dekrementiert werden.

Wird ein JSR (IND)-Befehl simuliert, muß nach wie vor die Rücksprungadresse auf den Stapel gelegt werden. Dies würde entfallen, wenn die Rücksprungadresse direkt auf »SPRG JMP 0000« folgen und der JMP-Befehl bei SPRG in JSR umgewandelt würde.

Damit soll das Thema »Selbstmodifikation« abgeschlossen sein. Die vorgestellten Programmiertechniken bieten fast unbegrenzte Möglichkeiten, hier konnte ich nur einen

kleinen Überblick geben, welcher aber für fortgeschrittene Programmierer ausreicht.

11. Mehr über relative Adressierung

So wie wir schon die Tücken der Zeropage-Adressierung zumindest teilweise beseitigen konnten, wollen wir uns mit der in vergleichbarer Weise leistungsstarken Relativ-Adressierung auseinandersetzen.

a) So vermeidet man JMP

Oft muß eine Stelle im Programm angesprungen werden, ohne daß erst eine Bedingung geprüft wird. Diese Stelle ist nicht selten weniger als 128 Byte vom Sprungbefehl entfernt, könnte also relativ adressiert werden.

Daher ist es in vielen Fällen möglich, einen Branch-Befehl - obwohl diese Befehle eine Bedingung (C=0..) prüfen - zu verwenden.

Beispiel:

```
7050    BNE 7040
7052    JMP 708A
```

Kann ersetzt werden durch:

```
7050    BNE 7040
7052    BEQ 708A
```

da bei 7052 in jedem Fall das Z-Flag = 0 ist (dafür sorgt der Abfang-Befehl BNE) und somit immer verzweigt wird.

Man könnte den BEQ-Befehl als »Pseudo-Verzweigungsbefehl« bezeichnen, da die Bedingung gar nicht überprüft werden müßte (sie ist sowieso erfüllt).

Der Branch-Befehl übertrifft den JMP-Befehl deutlich an Effektivität, da ein Byte weniger verbraucht wird.

Im übrigen ist auch bei

```
7050    BVS 7040
7052    CLV
```

der CLV-Befehl überflüssig, solange vor 7052 der Befehl von 7050 verarbeitet wird.

b) Zugriff auf Befehle in »Umgebung«

Unter »Umgebung« wollen wir den Bereich um einen Programmteil verstehen, der über relative Adressierung angesprochen werden kann. Da in diesem oft ähnliche Befehlsfolgen stehen wie im anderen Programm, läßt sich hier durch gezielten Zugriff auf die »Umgebung« der Speicherplatzbedarf senken.

Beispielsweise stehen an vielen Stellen im Programm RTS-Befehle. Diese werden, wenn ein Unterprogramm verlassen werden soll, manchmal durch einen Branch-Befehl angesprungen.

```
X1      RTS       ; Ende eines im Speicher voraus-
                    gehenden Unterprogramms
UP      ......    ; Unterprogramm
TEST    BEQ X2    ; Unterprogramm verlassen, falls
                    Z=0
        ......    ; andernfalls weiteres Programm
X2      RTS       ; Ende des Unterprogramms
```

Wenn X1 von TEST aus relativ adressiert werden kann, können wir folgendermaßen ein Byte sparen:

```
X1      RTS
UP      ......
TEST    BEQ X1    ; nach X1 springen, wo auch ein RTS
                    steht
        ......
X2      RTS         wird nicht mehr benötigt
```

Noch ein Beispiel aus dem Basic-Interpreter. Bei Adresse $AF08 stehen zwei Befehle, die einen »SYNTAX ERROR« erzeugen.

Nun gibt es im Basic-Interpreter unzählige Stellen, an denen ein »SYNTAX ERROR« aufgerufen werden muß. Deshalb steht dort nur »JMP $AF08«. Diese Stellen werden bei Bedarf relativ adressiert, so daß nicht an jeder Stelle, an der ein SYNTAX ERROR aufgerufen wird, der Befehl »JMP $AF08« stehen muß.

Zur Übung könnten Sie noch versuchen, im Programm Tabellen-Beispiel (Listing 11) die Menüroutine (insbesondere die Routinen HOME, DOWN, UP, EXEC), in der beispielsweise wiederholt STX MPT steht, durch Zugriff auf »Umgebung« zu optimieren. Besonders hilfreich dürfte es sein, zunächst statt Branch-Befehlen JMPs einzusetzen und dann zu überlegen, inwieweit die JMPs durch Branches ersetzt werden können, weil zum Beispiel nach »LDX #0« das Z-Flag immer gesetzt ist etc.

12. Puffer-Technik

In der Computerei fällt der Begriff »Puffer« sehr häufig. Beim C 64 gehören der Kassetten- und der Tastaturpuffer gemeinhin zu den bekanntesten Puffern. Statt »Puffer« kann man auch Zwischenspeicher sagen. Puffer dienen nämlich immer als Zwischenspeicher.

Zunächst wollen wir klären, was zu einem Puffer gehört.

a) Was benötigt ein Puffer?

- Pufferspeicher

Selbstverständlich muß ein Puffer einen bestimmten Speicherbereich belegen, in dem die Werte zwischengespeichert werden.

Ebenso muß die maximale Puffergröße festgelegt werden, damit geprüft werden kann, ob sich der Puffer schon angefüllt hat. Beim Kassettenzugriff werden vorerst alle Byte, die auf die Kassette sollen, im Puffer (ab $033C) zwischengespeichert. Ist dieser Puffer voll, würde er beim nächsten Byte, das er aufnehmen soll, überlaufen (das heißt, die maximale Puffergröße überschreiten). Deshalb wird dann Byte für Byte der Puffer entleert, indem die Bytes auf Kassette geschrieben werden. Jedes auf die Kassette geschriebene Byte belegt keinen Speicher mehr im Puffer, so daß der Puffer wieder aufnahmefähig ist.

Damit das Programm, das den Puffer verwaltet, auch weiß, aus welcher Adresse im Puffer es sich das nächste Byte holen soll beziehungsweise wo im Puffer das nächste Byte abgelegt werden soll, gibt es noch einen

- Pufferzeiger

Auf englisch heißt er »BUFFER-POINTER«, woher auch die Abkürzung »B-P« beim Floppy-Befehl zur Manipulation des Pufferzeigers stammt.

Dieser Pufferzeiger kann mit dem Stapelzeiger verglichen werden. Auf keinen Fall ist er mit dem

- Puffervektor

zu verwechseln, der die Startadresse des Pufferspeichers beinhaltet. Ein Puffervektor ist nicht unbedingt erforderlich, erhöht aber die Flexibilität.

Damit wären die Fachausdrücke im Zusammenhang mit Puffern geklärt.

b) Wann verwendet man Puffer?

Puffer dienen in der Regel als Zwischenspeicher, wie zum Beispiel der Basic-Eingabepuffer (ab $0200).

Im Fall des Tastatur- oder Diskettenpuffers aber sind die Puffer als Verbindungsstelle zwischen zwei parallel arbeitenden Programmen beziehungsweise Peripherie-Geräten vorgesehen (interruptgesteuerte Tastaturabfrage/Hauptprogramm im Computer, DOS/Betriebssystem des Computers).

Die Puffer sind in diesen Fällen Bereiche, auf die zwei (quasi-) parallel arbeitende Programme zugreifen.

Bei Computern, die ein wirklich starkes Multitasking bieten (wie der Commodore Amiga) finden Puffer weitaus mehr Verwendung als beim C 64, der nur einen quasiparallelen Ablauf ermöglicht.

Daher werden bei ihm Puffer hauptsächlich im I/O-Bereich verwendet, zum Beispiel bei Druckern, Datasette, Floppy, Tastatur etc. (I/O = Input/Output = Eingabe/Ausgabe).

13. Pass-Technik

a) Begriffserläuterung

Der Begriff »Pass« wurde schon mehrfach im 64'er-Magazin erläutert (unter anderem Ausgabe 7/85, Seite 51).

Am einfachsten kann der Begriff als »Schritt beim Programmmablauf« verstanden werden. Mit »Schritt« ist hier nicht ein einzelner Befehl, sondern ein größerer Block im Programm gemeint.

Wenn ein Programm in drei Passes (Durchläufen) arbeitet, heißt dies, daß drei Schleifen hintereinander abgearbeitet werden, die alle eine Teilaufgabe erfüllen, die erst in Verbindung mit den anderen Passes eine größere Aufgabe (zum Beispiel eine Assemblierung) ausfüllen kann. Jeder einzelne Pass führt eine bestimmte Tätigkeit aus, die für das Funktionieren der darauffolgenden Passes unbedingt erforderlich ist. Pass 1 wirkt also wie eine Initialisierung von Pass 2 etc.

Komplexe Programme in Schritte (Passes) zu gliedern, gehört zu den Grundregeln des strukturierten Programmierens.

b) Beispiele von Anwendungen der Pass-Technik

Besonders umfangreiche Programme wie Assembler (Hypra-Ass), Compiler (Austro-Speed) und Interpreter (z.B. Comal) sind immer in mehrere Passes eingeteilt.

So erfolgt bei den meisten Assemblern im ersten Pass ein Syntax-Check und das Anlegen der Symbol-Tabelle. Erst im zweiten Pass wird der Objektcode generiert, wobei die bereits erstellte Symboltabelle benötigt wird.

14. Diverse Tips zur optimalen Speichernutzung

Mit übermäßig viel RAM ist der C 64 bekanntlich nicht gesegnet. Bei vielen Anwendungen braucht man auch das letzte Byte.

Sie werden nun mehrere Tips erhalten, wie man den wenigen vorhandenen Speicher möglichst sparsam verwenden kann.

Zu den speicherplatzaufwendigsten Einrichtungen gehören die Puffer. Der Kassettenpuffer beispielsweise belegt den RAM-Bereich $033C - $03FB, auf den man somit oft verzichten muß.

Hier wollen wir einfach den Kassettenpuffer in den Bildschirmspeicher (ab $0400 in Normaleinstellung) verlegen.

```
LDA #<$400
LDY #>$400
STA $B2
STY $B3
```

Da der Bildschirm beim Kassettenbetrieb ohnehin abgeschaltet wird, fällt dies nicht auf. Nach dem Kassettenbetrieb sollte man aber den Bildschirm unverzüglich löschen.

Ebenso kann man andere Puffer, für die es einen Vektor gibt, problemlos nach $400 verlegen, sofern sie nicht größer als 1000 Byte sind.

Ein Problem für sich stellt das RAM ab $E000 (also unter dem Betriebssystem!) dar. Diesen Speicher kann man nur durch Bank-Switching nutzen, wobei man noch auf das Betriebssystem verzichten muß, solange der $E000-Bereich auf RAM geschaltet ist.

Hier können wir uns zunutze machen, daß der VIC auch ohne Ändern des Prozessor-Ports (Adresse $0001) auf diesen RAM-Bereich zugreifen kann. Für Grafikbilder oder einen geänderten Zeichensatz ist der $E000-Bereich bestens geeignet.

Oft wird der $E000-Bereich zur Ablage verschiedener Daten verwendet, auf die nicht andauernd zugegriffen werden muß.

Man könnte aber auch das Betriebssystem ins RAM ab $E000 kopieren und diejenigen Bereiche, in denen nicht benötigte Routinen stehen (zum Beispiel für Kassettenbetrieb) einfach überschreiben. Dies ist dann sinnvoll, wenn nur ein paar Byte im $E000-Bereich gebraucht werden. Außerdem ist eine gute Kenntnis des C 64-ROMs erforderlich.

Nun wollen wir noch besprechen, wie der Speicherplatzbedarf eines Programms niedriggehalten werden kann. Dazu wurde im Laufe des Kurses schon einiges gesagt (Unterprogramme statt Makros verwenden etc.).

Jedes Programm benötigt eine Menge Flags. Meist belegt ein Flag genau 1 Byte, für dessen Inhalt es oft nur zwei mögliche Werte gibt: einen für »JA« und einen für »NEIN«.

Für diese primitive Unterscheidungsform genügt aber auch 1/8 Byte, also ein Bit.

Wenn Sie sich die Register des Video-Clip ansehen, werden Sie feststellen, daß fast jedes VIC-Register mehrere Funktionen hat, weil jedem Bit eine eigene Bedeutung zukommt. Würde der VIC hier statt auf Bits auf Bytes zugreifen müssen, wäre er

1. langsamer und
2. würde der Speicherplatzaufwand für die Register sich vervielfachen.

Man sollte also bei Flags jedem Bit eine Bedeutung geben und nur die Bits prüfen:

```
BIT FLAG
```

Danach ist das N-Flag gesetzt, falls das 7. Bit im FLAG gesetzt ist, und das V-Flag, falls das 6. Bit gesetzt ist. Die übrigen Flags erhält man über das Z-Flag im Prozessor-Status-Register mit Hilfe des Akkus. Angenommen, man möchte testen, ob Bit 0 im Flag gesetzt ist oder nicht, dann macht das folgendes Programm:

```
LDA #01
BIT Flag
BNE ???          ; (Bit gesetzt)
.
.
.
; (Bit nicht gesetzt)
```

Der Bit-Befehl »ANDet« den Inhalt des Akkus mit dem Inhalt der Speicherzelle »Flag«. Möchte man Bit 1 testen, so ist der Befehl LDA #01 zu ersetzen durch LDA #02 und so weiter.

Durch Selbstmodifikation können Flags bekanntlich vermieden werden. Aber auch sonst bietet die Selbstmodifikation die Möglichkeit, Speicherplatz zu sparen: die Steuerung einer Sprungtabelle belegt mit Selbstmodifikation weniger Speicher als ohne.

Auch die »Wegwerfmethode« ist sehr vorteilhaft; Programmteile werden einmal abgearbeitet und dann (zum Beispiel durch Nachladen) überschrieben.

Damit hätten wir unseren Kurs abgeschlossen. Ich hoffe, daß er Ihnen etwas Spaß gemacht hat und Sie einige interessante Informationen herausholen konnten. Sie sollten sich jedoch darüber im klaren sein, daß einige der hier vorgestellten Methoden die Lesbarkeit eines Assembler-Listings einschränken können. Also, verzichten Sie, wenn nicht unbedingt notwendig, auf allzu trickreiche Programmierung. Falls Sie noch Fragen oder Probleme haben (vielleicht erst wegen diesem Artikel), dann schreiben Sie doch einfach. (Florian Müller/rs)

3 DI BRAUCHT DER P

1. Hypra-Ass

Ein Makro-Assembler der Spitzenklasse. Er erlaubt es, Maschinensprache-Programme ähnlich kom-komfortabel wie in Basic zu schreiben. Durch Makros – dies sind immer wieder benötigte Unterprogramme, die über ihren Namen aufgerufen werden – und bedingte Assemblierung durch logische Verzweigungen ist große Übersichtlichkeit auch bei langen Programmen gewährleistet. Ein weiterer Vorteil ist der Editor des Hypra-Ass, der durch eine formatierende LIST-Routine größtmögliche Übersicht am Bildschirm gewährleistet.

Theorie ist nicht alles. Mächtige Programmierwerkzeuge und Tools erst versetzen Sie in die Lage, Ihre Fähigkeiten in der Assemblerprogrammierung richtig umzusetzen. Diese wollen wir Ihnen auf den nächsten Seiten anbieten. Es sind in der Praxis tausendfach bewährte und erprobte Programme:

NGE ROGRAMMIERER

2. Der SMON
Ein leistungsfähiger Speichermonitor, der es Ihnen erlaubt, bis in die tiefsten Tiefen Ihres Computers vorzudringen. Mit diesem Werkzeug lassen sich die Prozessorregister anzeigen und beeinflussen, Speicherbereiche anzeigen, vergleichen, verschieben und und und. Was ebenfalls nicht fehlt, ist eine Funktion zum Disassemblieren, also der Entschlüsselung des Maschinencodes, und umgekehrt ein Miniassembler, der es Ihnen erlaubt, »mal eben« ein kleines Programm einzugeben. Der Trumpf des SMON ist der integrierte Diskettenmonitor, der Ihnen auch die volle Kontrolle über die Floppystation gibt.

3. Der REASS
Quasi die Umkehrung des Hypra-Ass und mindestens genauso wichtig. Sie haben beispielsweise ein Maschinenprogramm bekommen, das Ihnen ganz gut gefällt, bis eben auf einige Kleinigkeiten, die Sie ändern wollen. Doch wie? Im Dickicht der reinen Maschinencodes findet sich niemand zurecht und ein Quellcode ist nicht vorhanden. Hier hilft der Reass: Er macht aus dem Maschinenprogramm wieder gut lesbaren und strukturierten Quellcode, der mit Labels und Zeilennummern versehen direkt mit dem Hypra-Ass bearbeitet werden kann.

1 HYPRA

Machen Sie es wie die Profis. Schreiben Sie Programme in Maschinensprache. Dieser leistungsstarke Makro-Assembler macht es möglich.

Hypra-Ass (Listing 1) ist ein in Maschinensprache geschriebener Drei-Pass-Makroassembler mit integriertem Editor für den C 64. Er wird mit

```
LOAD "HYPRA-ASS",8
```

geladen und durch RUN gestartet. Nach dem Start meldet sich Hypra-Ass mit »break in 0« und »ready«. Alle Basic-Befehle sind nach dem Start noch zu verwenden, bis auf die Befehle LET und FOR, die Variable anlegen. Der Befehl RUN dient jetzt zum Starten der Assemblierung.

Der Quelltext

Der Quelltext wird vom Hypra-Ass-Editor in Basic-Programmzeilen abgelegt. Soweit wie möglich werden unnötige Blanks dabei eliminiert. Für die einzelnen Quelltextzeilen gelten die folgenden Vereinbarungen:

1. Bei der Eingabe einer Zeile wird hinter der Zeilennummer ein Minuszeichen eingegeben.
2. Jede Zeile enthält höchstens einen Assemblerbefehl.
3. Vor einem Assemblerbefehl darf in derselben Zeile höchstens ein Label stehen.
4. Label beginnen direkt hinter dem Minuszeichen.
5. Vor jedem Assemblerbefehl steht mindestens ein Blank.
6. Label und Assemblerbefehl werden durch mindestens ein Blank voneinander getrennt.
9. Reine Kommentarzeilen müssen als erstes Zeichen hinter dem Minuszeichen ein Semikolon haben.
10. Pseudo-Opcodes (.ba, .eq...) können direkt hinter dem Minuszeichen beginnen. Beispiele:

```
100  -.ba $C000
110  -initialisierung
120  -; reine Kommentarzeile
130  -              lda $14; Kommentar hinter
                             einem Befehl
140  -marke         ldx $15; mit Label davor
```

Zur bequemeren Eingabe und Bearbeitung des Quelltextes stellt Hypra-Ass im Editor insgesamt 25 Befehle zur Verfügung (Tabelle 1).

Rechnungen im Quelltext

Hypra-Ass erlaubt vier Grundrechenarten plus Potenzierung, die logischen Operationen NOT, AND und OR, die Vergleiche »gleich«, »kleiner« und »größer« sowie den Einsatz der Funktionen <(...) und >(...), die das Low- beziehungsweise Highbyte eines Argumentes liefern. Die logischen Operatoren und die Vergleiche werden so abgekürzt:

!n! = not
!a! = and
!o! = or
!=! = gleich
!<! = kleiner als
!>! = größer als

Das Ergebnis eines Vergleiches ist —1, falls wahr, 0, falls nicht wahr: (1!=!2)=0 (1!=!1)=—1

Auch die NOT-Verknüpfung arbeitet wie in Basic: !n!1 = —2.

Das Argument in den Low-/Highbyte-Funktionen muß im Bereich 0 ≤ Argument ≤65535 liegen.

Außer Dezimalzahlen sind Hex-Zahlen erlaubt, die durch ein vorangestelltes Dollarzeichen kenntlich gemacht werden:

$C000 = 49152 $10 = 16 $a = 10...

Die Hexzahlen können auch in den Basic-Befehlen verwendet werden.

Hypra-Ass-Variable (Label)

Der Wert einer Hypra-Ass-Variablen kann zwischen 0 und $FFFF liegen. Variablennamen können beliebig lang sein, wobei das erste Zeichen des Variablennamens ein Buchstabe sein muß. Weitere Zeichen können Buchstaben, Ziffern oder das Hochkomma sein. Alle Zeichen des Namens sind signifikant.

Im Zusammenhang mit der Verwendung von Makros muß zwischen globalen und lokalen Variablen unterschieden werden. Jede Variable erhält beim Anlegen eine sogenannte Ordnungszahl, die angibt, im wievielten Makroaufruf das Anlegen stattfand. Befindet man sich in gar keinem Makro, ist die Ordnungszahl entsprechend Null.

Variable mit unterschiedlicher Ordnungszahl sind trotz

gleichen Namens nicht gleich. Man kann also davon sprechen, daß Variable gleicher Ordnungszahl lokal sind.

Die Konstruktion mittels Ordnungszahlen dient dazu, Fehler durch doppelte Benutzung von Labeln bei mehrmaligem Aufruf von Makros zu verhindern.

Andererseits sind aus einem Makro »heraus gesehen« alle Variablen mit anderer Ordnungszahl als im Makro selbst »unsichtbar«. Um aber bequem Makros in Makros aufrufen und bequem Label verwenden zu können, die in mehreren Makros benutzt werden sollen (etwa Betriebssystemroutinen), gibt es die globalen Variablen.

Globale Variable sind, wie der Name schon verrät, im Gegensatz zu den lokalen Variablen unabhängig von der Ordnungszahl überall definiert.

Alle Makronamen sind per Definition global.

Alle Variablen sind bei Hypra-Ass redefinierbar gehalten, das heißt alle Variablen können durch eine Wertzuweisung jederzeit verändert werden.

Eine doppelte Benutzung von Labeln vor Assemblerbefehlen wird jedoch durch einen »Label twice«-Error (Tabelle 2) geahndet, da dies zu einem falschen Ergebnis der Assemblierung führen würde.

Die Makros von Hypra-Ass

Makros sind meist kürzere Befehlsfolgen, die im Quelltext häufiger vorkommen, und deshalb unter einem Makro zusammengefaßt werden. Zu jedem Makro gehört ein Makroname, mit dem es aufgerufen werden kann. An jedes

-ASS – Ein Assembler der Spitzenklasse

Hypra-Ass-Makro können beliebig viele Parameter übergeben werden, deren aktueller Wert dann bei der Assemblierung im Makro eingesetzt wird. Makros können bei Hypra-Ass an beliebiger Stelle im Quelltext definiert werden. Alle Makronamen sind global, alle Parameter und makrointernen Label sind lokal. Das heißt verschiedene Makros können durchaus Label beziehungsweise Parameter gleichen Namens verwenden.

Ein Beispiel für ein einfaches Makro:

Es wird immer wieder die Befehlsfolge benötigt, Akkumulator und X-Register mit dem Inhalt zweier aufeinanderfolgender Speicherzellen zu laden. Ein Makro dazu könnte folgendermaßen aussehen:

```
100  -.ma ldax (adresse)
110  -       lda adresse
120  -       ldx adresse+1
130  -.rt
```

Der .ma-Pseudobefehl wird gefolgt von einem Variablennamen, dem Makronamen, und einer Parameterliste in runden Klammern, falls Parameter vorhanden sind. Hier ist es ein Parameter, die Adresse der Speicherzelle, die in den Akku soll. Sind mehrere Parameter vorhanden, werden sie durch Kommata getrennt. In die Parameter setzt der Assembler bei jedem Aufruf den aktuellen Wert, der im Aufruf steht. Rufe ich also ldax (2) auf, so entsteht bei der Assemblierung des Makros die Folge lda 2, ldx3, entsprechend führt der Aufruf mit ldax (label) zu lda label, ldx label+1.

Die Parameterliste darf in der Definitionszeile eines Makros nur aus einer Folge von Variablennamen bestehen, während im Aufruf als aktuelle Parameter beliebige Ausdrücke erlaubt sind. Hinter der Definitionszeile mit dem .ma-Pseudo folgt der eigentliche Makroinhalt, also das, was bei einem Aufruf des Makros assembliert werden soll.

Natürlich sind hier nicht nur einfache Befehle wie im Beispiel gestattet. Genausogut können im Makro Verzweigungen und Sprünge ausgeführt werden, es kann bedingt assembliert werden und weitere Makros können aufgerufen werden. Für die Schachtelung von Makros besteht keine Grenze außer der Fassungskapazität des Prozessorstacks.

Als Beispiel: Wird ein Makro mit zehn internen Labeln 100mal aufgerufen, ergibt sich schon für die dadurch erzeugten lokalen Label ein Platzbedarf von genau 7000 Byte.

Sollte irgendwann der Fall eintreten, daß Label und Quelltext zusammen nicht mehr ins RAM passen, erhalten Sie den »too many labels«-Error (Bild 2). Dies ist allerdings mehr ein theoretischer Fall, denn auch bei der Assemblierung von Hypra-Ass selbst wurden trotz extensiver Benutzung von Labels nicht einmal 500 gebraucht. Sie können aber davon ausgehen, daß Ihnen immer mindestens Platz für 1 170 Label zur Verfügung steht — in den allermeisten Fällen sogar erheblich mehr.

Selbstaufrufe von Makros sind auch nicht verboten. Inwieweit eine solche Konstruktion überhaupt sinnvoll sein kann, bleibt jedem selbst zu prüfen.

Zurück zur Makrodefinition: Jede Makrodefinition muß unbedingt mit dem Pseudo .rt (return) abgeschlossen sein. Trifft der Assembler bei der Abarbeitung eines Makros auf .rt, so heißt das für ihn, die Assemblierung hinter dem Aufruf fortzusetzen.

Vor der .ma und .rt-Anweisung dürfen in derselben Zeile keine Label stehen. Die Makrodefinition selbst wird in Pass 1 und Pass 2 überlesen. Es zählen also nur die Makroaufrufe bei der Assemblierung.

Der Aufruf eines Makros erfolgt durch den Pseudobefehl ..., gefolgt vom Makronamen und der aktuellen Parameterliste in runden Klammern.

Wertzuweisung an Label

Zwei Pseudobefehle stehen zur Verfügung, um Label einen Wert zuzuweisen:

.eq – weist einem Label einen Wert zu, ohne die Ordnungszahl des Labels dabei zu verändern.

.gl – erklärt gleichzeitig das Label als global.

Beide Pseudos werden der eigentlichen Wertzuweisung vorangestellt, so wie LET in Basic:

```
100  -.eq marke = $FFC0
110  -.gl label = $200
```

Bei der Wertzuweisung an Label ist immer der Bereich einzuhalten, in dem ein Labelwert liegen darf (0 bis $FFFF).

Einfügen von Tabellen und Text

Drei Pseudo-Ops erleichtern das Einfügen von Tabellen und Text in den Quelltext. Dies sind:

.by – erlaubt das Einfügen von Bytewerten (Werten zwischen 0 und $FF). Einzelne Bytewerte werden durch Kommata voneinander getrennt. Auch Strings der Länge 1 sind als Bytewerte erlaubt. Beispiel:
100 -.by 0, "a", 123, "x", $fa

.wo – erlaubt das Einfügen von Adressen (Werten zwischen 0 und $FFFF). Mehrere Adressen werden durch Kommata voneinander getrennt. Die Adressen werden in der Folge Low/Highbyte in den Objektcode aufgenommen. Beispiel:
100 -.wo marke-1, label*2-1

.tx – erlaubt das Einfügen von Text in den Quelltext. Die einzelnen Zeichen des Textes werden als ASCII-Code im Objektcode aufgenommen. Beispiel:
100 -.tx "beispieltext"

Überall im Quelltext, wo Bytewerte erwartet werden, etwa bei der unmittelbaren Adressierung, können Strings der Länge 1 verwendet werden. Ein Befehl lda # " ist also erlaubt.

Die bedingte Assemblierung

Zur Unterstützung der bedingten Assemblierung bietet Hypra-Ass ein IF/ELSE/ENDIF-Konstrukt und ein IF/THEN-Konstrukt. Außerdem steht ein unbedingter Sprungbefehl zur Verfügung.

.on – entspricht dem IF/THEN von Basic. Hinter .on folgt ein Ausdruck, ein Komma und ein zweiter Ausdruck. Ist der erste Ausdruck wahr, wird zu der Zeilennummer gesprungen, die der zweite Ausdruck angibt. Beispiel:
`100 -.on switch !=! 7, 400`

Es wird die Assemblierung in Zeile 400 fortgesetzt, wenn switch gleich 7 ist.

.go – ergibt einen unbedingten Sprung zu der Zeile, die der Ausdruck hinter .go angibt. Beispiel:
`100 -.go 1000`

.if – wird gefolgt von einem Ausdruck. Ist der Ausdruck wahr, wird die Assemblierung hinter der .if-Zeile fortgesetzt, bis

.el – gefunden wird. Daraufhin wird

.ei – gesucht und dahinter die Assemblierung fortgesetzt.

Entsprechend erfolgt die Assemblierung von .el bis .ei, falls der Ausdruck hinter .if falsch ist. .el kann auch fehlen, es wird dann direkt hinter .ei fortgefahren.

/a 100,10	Automatische Zeilennumerierung. Hier mit der Startnummer 100 und der Schrittweite 10. Die automatische Zeilennumerierung wird ausgeschaltet, indem man direkt hinter dem ausgegebenen Minuszeichen RETURN eingibt.
/o	Re-New eines Quelltextes, der mit NEW gelöscht wurde, falls der Text nicht anderweitig zerstört wurde.
/d ; /d 100 ; /d —100 ; /d 100— ; /d 100-200	Löschen von Zeilen und Zeilenbereichen. Auch für das Löschen einzelner Zeilen sollte man den /d-Befehl verwenden, da man das Minuszeichen hinter der Zeilennummer doch immer wieder vergißt.
/e ; /e 100 ; /e —100 ; /e 100— ; /e 100-200	Formatiertes Listen von Zeilen und Zeilenbereichen. Label, Assembler-Befehle werden gemäß den Tabulatoren übersichtlich untereinander geschrieben.
/t 0,13 ; /t 1,24 ; /t 2,0 ; /t 3,10	setzt die Tabulatoren T0, T1, T2, T3 T0 = Tabulator für Assemblerbefehle T1 = Tabulator für den Kommentar bei der formatierten Ausgabe T2 = Tabulator für die Anzahl der Blanks, die am Anfang einer Ausgabezeile ausgegeben werden T3= Tabulator für die Symboltabelle
/x	Verlassen des Assemblers. Beim Verlassen des Programms wird ein Reset durchgeführt.
/p 1,100,200	Setzen eines Arbeitsbereichs (Page). Hier Bereich 1 von Zeile 100 bis 200, beide einschließlich. Bis zu 30 solcher Arbeitsbereiche sind erlaubt. Die Parameter der Arbeitsbereiche werden im Kassettenpuffer abgelegt.
/ziffer(n)	Formatiertes Listen der Page.
/n 1,100,10	Neu Durchnumerieren einer Page mit Startnummer und Schrittweite.
/f 1,"string"	Suchen einer Zeichenkette in einer Page. Dabei sind im String Fragezeichen als Joker erlaubt. Das Fragezeichen ersetzt ein beliebiges Zeichen. Zu beachten ist jedoch, daß im Quelltext unnötige Blanks entfernt wurden, wie ein Vergleich mit den Befehlen /e und LIST zeigt.
/r 1,"string1", "string2"	Ersetzen von Zeichenketten. String 2 darf nicht leer sein. Überall in der Page wird die Zeichenkette aus String 2 durch die aus String1 ersetzt. Auch beim Ersetzen ist in String 2 das Fragezeichen als Joker erlaubt. Da String 1 leer sein darf, können mit diesem Befehl auch Zeichenketten gelöscht werden.
/u 9000	Setzen des Quelltextstartes (Programmstartes). Normalerweise ist als Startwert die Adresse 7000 eingestellt. Durch Hochlegen des Startes kann man zum Beispiel einen Monitor in dem nun freien Bereich unterbringen.
/b	Anzeige der aktuellen Speicherkonfiguration. Es wird angezeigt: a) der normale Quelltextstart 7000 als Merkhilfe b) der aktuelle Quelltextstart c) das Quelltextende d) die Anzahl der noch verbleibenden Bytes für den Quelltext
/l"name" ; /s"name" ; /v"name" ; /m"name"	Kurzform der Befehle LOAD, SAVE, VERIFY und MERGE
/g 8	Die zugehörige Gerätenummer kann mit diesem Befehl eingestellt werden. Voreingestellt ist das Gerät 8.

Zur Unterstützung des Umgangs mit dem Floppy-Laufwerk 1541 sind drei Befehle implementiert:

/i	— Lesen des Inhaltsverzeichnisses von Floppy ohne Verlust des geschriebenen Quelltextes
/k	— Lesen des Fehlerkanals
/@	— Übermittlung von Befehlen an die Floopy

Diese drei Befehle entsprechen denen des DOS 5.1.

Auch zur Farbgebung des Bildschirms sind zwei Befehle vorhanden, die die Hintergrund- und die Rahmenfarbe setzen.

/ch 0	— Setzen der Hintergrundfarbe
/cr 0	— Setzen der Rahmenfarbe

Nach erfolgter Assemblierung kann nun die erzeugte Symboltabelle mit zwei Befehlen ausgegeben werden:

/!	— Ausgabe in unsortierter Form
/!!	— Ausgabe sortiert

Es werden nur Label ausgegeben, die entweder global oder von der Ordnung Null sind.

Beide Dumps können mit der CTRL-Taste verlangsamt und mit der STOP-Taste angehalten werden.

Mit OPEN... und CMD... können die Dumps an andere Geräte gesendet werden.

Als Ergänzung zum Basic-Befehl PRINT, der aufgrund der Tokenbildung nicht alle Labelnamen verarbeiten kann, kann der Befehl ← verwendet werden.

Basic-Funktionen wie PEEK sind nur über den PRINT-Befehl erreichbar. Die Funktionen <(...) und >(...) sind außerhalb des Quelltextes nur durch ← zu verwenden. Mit dem ←-Befehl kann genau wie im Quelltext gerechnet werden.

Tabelle 1. Die Editorbefehle von »Hypra-Ass«

Zusätzlich zu den Fehlermeldungen, die von Interpreterroutinen wie »illegal quantity« oder »syntax« stammen, gibt Hypra-Ass folgende Meldungen aus:
1. **can't number term** — ein Ausdruck kann von Hypra-Ass nicht berechnet werden. Möglicher Grund kann die falsche Abkürzung eines Operators sein.
2. **end of line expected** — bei der Abarbeitung einer Zeile wurde statt des Zeilenendes etwas anderes gefunden.
3. **no mnemonic** — ein Mnemonic kann nicht identifiziert werden.
4. **unknown pseudo** — ein Pseudo-Op wurde falsch abgekürzt.
5. **illegal register** — ein Assemblerbefehl existiert in der gewählten Adressierungsart nicht mit dem gewählten Register.
6. **wrong address** — ein Assemblerbefehl existiert nicht in der gewählten Adressierungsart.
7. **illegal label** — das erste Zeichen eines Labels war kein Buchstabe.
8. **unknown label** — in Pass 2 wurde ein unbekannter Labelname entdeckt
9. **branch too far** — eine Verzweigung führt über eine zu große Distanz.
10. **label declared twice** — ein Labelname wurde zweimal benutzt.
11. **too many labels** — Label und Quelltext passen zusammen nicht mehr in den Speicher.
12. **no macro to close** — die Anzahl der .ma-Anweisungen stimmt nicht mit der Anzahl der .rt-Anweisungen überein.
13. **parameter** — im Makroaufruf stimmt die Parameterliste nicht mit der Parameterliste der Definition überein.
14. **return** — es liegt keine Rückkehradresse auf dem Stack, als eine .rt-Anweisung ausgeführt werden sollte.

Hinzuweisen ist noch auf eine einfache Möglichkeit, den »label twice-error« zu vermeiden:
Legt man eine Makrodefinition um einen beliebigen Block des Quelltextes, so sind alle Label in dem Block automatisch lokal. Auf diese Weise kann schon vorhandener Quelltext in neuen eingefügt werden, ohne daß man sich um doppelt verwendete Labelnamen kümmern muß.

Tabelle 2. Fehlermeldungen von »Hypra-Ass«

Auf eine Schachtelung von IF-Konstrukten wurde wegen des Zwecks der bedingten Assemblierung verzichtet. Beispiel:

```
100  -.if switch !=! 6
110  -       lda#0
120  -.el
130  -       lda#2
140  -.ei
```

Wenn switch gleich 6 ist, erhält man lda#0, sonst wird lda#2 erzeugt. Vor den Pseudos .if, .el und .ei dürfen keine Label in derselben Zeile stehen.

Verkettung von Quelltexten

Mit dem Pseudo .ap (apend) kann ein weiterer Quelltext am Ende des Pass 2 automatisch nachgeladen werden, wobei der Programmzähler aus der vorangegangenen Assemblierung erhalten bleibt.

Hinter .ap muß der Name des nachzuladenden Files in Anführungszeichen stehen.

Eine Besonderheit von Hypra-Ass bildet im Zusammenhang mit verketteten Quelltexten der Pseudo-Opcode .co (common).

Dieser Befehl bewirkt zunächst, daß alle Variablen/Label, die hinter der .co-Anweisung in einer Liste stehen, an den nachgeladenen Teil übergeben werden.

Zweitens bleiben alle Quelltextzeilen bis zur common-Zeile beim Nachladen erhalten. Steht also etwa ein Makro vor der common-Zeile, wird auch das Makro übergeben. Zu beachten ist dabei:

a) Es sollten keine Makroaufrufe im common-Bereich stehen, es sei denn innerhalb eines Makros.
b) Die .ba-Anweisung, die die Startadresse des Objektcodes bestimmt, sollte außerhalb des common-Bereiches liegen, damit nach dem Nachladen nicht wieder mit der gleichen Startadresse assembliert wird.
c) Wertzuweisungen an Label sollten ebenfalls außerhalb des common-Bereiches liegen, um Platz für den nachgeladenen Quelltext zu gewinnen.

Direktes Senden des Objektcodes zur Floppy

Der Pseudobefehl .ob (object), gefolgt vom Filenamen ,p,w in Anführungszeichen, sendet den erzeugten Objektcode direkt zur Floppy.

Geschlossen wird das so erzeugte Objektfile durch den Pseudobefehl .en.

Sollte während der Assemblierung ein Fehler entdeckt werden und das Objektfile nicht schon durch die Hypra-Ass-Fehlerroutine geschlossen worden sein, geben Sie bitte CLOSE 14 ein.

Ausgabe von formatierten Listings

1) **.li 1,3,0** sendet ein formatiertes Listing des Quelltextes unter der logischen Filenummer 1 an das Gerät 3 mit der Sekundäradresse 0 (Bildschirm). Die Parameter hinter .li entsprechen denen des OPEN-Befehls. So ist es auch möglich, mit .li 2,8,2, "test,u,w" das Listing auf eine Userdatei zu leiten und so weiter.

Der .li-Pseudobefehl muß der erste Befehl im Quelltext sein, wenn alle Zeilen gelistet werden sollen. Die Zeilen bis einschließlich .li werden nicht ausgegeben. Die gelisteten Zeilen haben folgendes Format:

```
c000 a0b0c0: 1000 -marke befehl ;kommentar
```

Die Steuerung der Formatierung erfolgt mit dem Editorbefehl /t. Bei Zeilen, die Pseudobefehle enthalten, wie .eq... werden keine Adressen und Opcodes ausgegeben.

2) **.sy 1,3,0** sendet am Ende von Pass 2 die sortierte Symboltabelle. Die Formatierung wird hier durch /t3,... gesteuert. Die Labelwerte werden hexadezimal ausgegeben.

Eine Zeile der Symboltabelle sieht dann folgendermaßen aus:

sprungziel = $ffd2

Das Listing des Quelltextes erhält die Kopfzeile »Hypra-Ass Assemblerlisting:«. Die Symboltabelle erhält die Kopfzeile »Symbols in alphabetical order«.

3) **.dp t0,t1,t2,t3** setzt aus dem Quelltext heraus die Tabulatoren
t0 = Tabulator für Assemblerbefehle
t1 = Tabulator für den Kommentar bei der formatierten Ausgabe
t2 = Tabulator für die Anzahl der Blanks, die am Anfang einer Ausgabezeile ausgegeben werden.
t3 = Tabulator für die Symboltabelle

4) **.st** beendet die Assemblierung

Nach dem zweiten Pass wird die Meldung »end of assembly« gefolgt von der Assemblierungsdauer in Minuten, Sekunden und Zehntelsekunden ausgegeben. Dahinter folgt die Zeile »base = $XXXX last byte at $YYYY«.

Eine Zusammenfassung aller Pseudobefehle finden Sie in Bild 3.

Eines der Ziele bei der Entwicklung von Hypra-Ass war es auch, die Editierung von Quelltexten möglichst bequem zu machen. Dazu wurden etliche Funktionen, die im normalen Basic-Editor stets gebraucht, aber nie vorhanden sind, in den Hypra-Ass-Editor eingebaut.

Hypra-Ass-Editor

Als Grundlage des Hypra-Ass-Editors blieb dabei der Basic-Editor erhalten.

Ein Hypra-Ass-Quelltext wird also im Prinzip genauso eingegeben wie ein Basic-Programm. **Allerdings muß hinter der Zeilennummer immer ein Minuszeichen eingegeben werden,** das den Beginn der Quelltextzeile bildet. So eingegebene Quelltextzeilen werden als ASCII-Zeilen in den Speicher übernommen. Alle überflüssigen Blanks werden entfernt.

Jede eingegebene Zeile wird sofort nach der Übernahme formatiert ausgegeben, um die Übersichtlichkeit des Quelltextes zu gewährleisten.

Im Gegensatz zum Basic-Editor kann unter Hypra-Ass eine Zeile nicht dadurch gelöscht werden, daß nur die Zeilennummer eingegeben und anschließend <RETURN> gedrückt wird. Bei Hypra-Ass ist unbedingt darauf zu achten, daß hinter der Zeilennummer ein Minuszeichen eingegeben wird. Drückt man nun die RETURN-Taste, ist die Zeile auch verschwunden. Da aber dieses Minuszeichen hinter der Zeilennummer meistens vergessen wird, ist es empfehlenswert, nicht nur Zeilenbereiche, sondern auch einzelne Zeilen mit dem Editor-Befehl »/D zeilennummer« zu löschen, bis auf die Zeile »0«, die sich mit dem »/D«-Befehl nicht löschen läßt. In diesem Fall geben Sie bitte »0-« <RETURN> ein.

Ein kleiner Fehler tritt beim Sortieren der Symboltabelle auf. Hypra-Ass stürzt ab, wenn die Symboltabelle genau 36, 73, 109 (und so weiter) Variablen oder Label enthält. Der Fehler liegt in den Speicherzellen $1EB8 bis $1EBB. Hier wurden zwei Branch-Befehle vertauscht. Es muß richtig lauten:

```
1EB8 90 D0    BCC 1EA8
1EBA D0 04    BNE 1EC0
```

Diese Änderung kann unmittelbar mit einem Monitor in die entsprechenden Speicherzellen geschrieben und anschließend gespeichert werden. Sollten Sie keinen Monitor haben, dann geben Sie bitte den folgenden Quelltext ein:

```
 10 -.BA $C000
;STARTADRESSE = $C000
 20 -                 LDY #0
 30 - LBL             LDA TAB,Y
;KORREKTUREN VORNEHMEN
 40 -                 STA $1EB8,Y
 50 -                 INY
 60 -                 CPY #4
 70 -                 BNE LBL
 80 -.EQ SOURCESTART = $1FD8 ;UND DIE KORRIGIERTE
 90 -.EQ NAMLEN = 12         ;VERSION
100-                  LDA #1  SPEICHERN
110-                  LDX #8
120-                  STA $FE
130-                  STX $FF
140-                  LDA #8
150-                  JSR $FFBA
160-                  LDA #NAMLEN
170-                  LDX #<(NAME)
180-                  LDY #>(NAME)
190-                  JSR $FFBD
200-                  LDA # $FE
210-                  LDX #<(SOURCESTART)
220-                  LDY #>(SOURCESTART)
230-                  JMP $FFD8
240-;
250-NAME              .TX "HYPRA-ASS.V1"
260-TAB               .BY $90,$D0,$D0,$04
```

Nach dem Assemblieren wird mit SYS 49152 <RETURN> Hypra-Ass geändert und unter dem neuen Namen »Hypra-Ass.V1« auf Diskette gespeichert.

1)	.ba $C000	gibt die Startadresse der Assemblierung an. Bei anderen Assemblern heißt dieser Befehl auch org oder * = .
2)	.eq label=wert	weist einem Label einen Wert zu
3)	.gl label=wert	weist einem globalen Label einen Wert zu
4)	.by 1,2,"a"	Einfügen von Byte-Werten in den Quelltext
5)	.wo 1234,label	Einfügen von Adressen in der Folge low/high
6)	.tx"text"	Einfügen von Text als ASCII-Werte
7)	.ap "file"	Verketten von Quelltexten
8)	.ob "file,p,w"	Senden des Objektcodes zur Floppy
9)	.en	Schließen des Objektfiles
10)	.on ausdruck,sprung	bedingter Sprung, wenn Ausdruck wahr
11)	.go sprung	unbedingter Sprung
12)	.if ausdruck	Fortführung der Assemblierung bei ELSE, falls Ausdruck falsch. Ansonsten hinter .if bis zu ELSE oder ENDIF.
13)	.el	Alternative zu den Zeilen, die hinter .if stehen
14)	.ei	Ende der IF-Konstruktion
15)	.co var1,var2	Übergabe von Labeln und Quelltext an nachgeladene Teile
16)	.ma makro (par1,par2)	Makrodefinitionszeile
17)	.rt	Ende der Makrodefinition
18)	...makro (par1,par2)	Makroaufruf
19)	.li, lfn, dn, ba	sendet formatiertes Listing unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
20)	.sy lfn, dn, ba	sendet formatierte Symboltabelle unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
21)	.st	beendet die Assemblierung
22)	.dp t0, t1, t2, t3	setzt die Tabulatoren T0, T1, T2, T3 aus dem Quelltext heraus

Vor den Anweisungen 12, 13, 14, 16 und 17 dürfen in derselben Zeile keine Label stehen.

Bild 3. Zusammenfassung aller Pseudobefehle

$0000	Zeropage
$033e	Bandpuffer als Zwischenspeicher
$0400	Video-RAM
$0800	Hypra-Ass
$1fd7	Raum für Quelltext und Label. Quelltext bis maximal $a000
$a000	Basic-Interpreter — darunter von c000 abwärts die Symboltabelle
$c000	frei!
$d000	I/O und so weiter
$c000	Kernal

Bild 4. Speicherbelegung von »Hypra-Ass«

Der »/A«-Befehl zur automatischen Zeilennumerierung reagiert recht sensibel. Wird mit diesem Befehl gearbeitet, darf der Cursor mit den entsprechenden Steuertasten auf keinen Fall auf eine andere Zeile gesetzt und <RETURN> gedrückt werden. Sollte das versehentlich doch passieren, läßt sich die so entstandene, etwas seltsam aussehende Zeile mit dem »/D«-Befehl problemlos löschen.

Diskettenbefehle können mit dem Editorbefehl »/@« zum Floppy-Laufwerk gesendet werden. Hinter den Editorbefehl werden die Diskettenbefehle angehängt. So informiert der Befehl »/@N:NEWDISK,ND« eine Diskette.

Eine feine Sache ist auch das Arbeiten mit dem »/P«-Befehl, der dazu dient, Arbeitsseiten beziehungsweise Arbeitsbereiche anzulegen. Durch diesen Befehl, auf den sich die meisten Editor-Befehle beziehen, ist es möglich, jedem zusammenhängenden Quelltextteil (Unterprogramme oder Unterprogrammblöcke) einen Arbeitsbereich zuzuordnen. Möchte man in der Page 3 etwas ändern oder nachschauen, LISTet der Befehl »/3« nur diesen Bereich und nicht das komplette Listing wie bei dem »/E«-Befehl. Legt man die einzelnen Arbeitsbereiche von vornherein so an, daß sie jeweils einen Zeilenbereich von zum Beispiel 5000 Zeilen überdecken, dürfte für die einzelnen Quelltextteile genügend Platz vorhanden sein, so daß beim Durchnumerieren der einzelnen Arbeitsbereiche keine Überlappungen auftreten können. Die Arbeitsbereiche selbst dürfen sich aber durchaus überlappen. So läßt sich zum Beispiel ein Arbeitsbereich von 0 bis 5000, ein zweiter von 10000 bis 15000 und ein dritter von 0 bis 15000 anlegen.

Bei dem Assembler selbst sind bisher keine Fehler bekannt. Deshalb möchte ich an dieser Stelle auf einige Dinge eingehen, mit denen viele Leser Schwierigkeiten hatten. Da wäre zum Beispiel das unmittelbare Erzeugen des Objektcodes auf Diskette mit dem ».OB«-Pseudo-Opcode.

Der Pseudo-Opcode ».OB "filename,P,W"« muß am Anfang des Quelltextes stehen und zwar in der ersten beziehungsweise zweiten Zeile (nach dem ».LI«-Pseudo zur Ausgabe des Assembler-Listings). Es ist auch möglich, den Objektcode und gleichzeitig das Assembler-Listing mit dem Befehl ».LI 2,8,2,"filename,U,W"« auf Diskette zu erzeugen. Der Grund ist der, daß zwei Kanäle zum Schreiben geöffnet werden müßten und das ist nicht möglich. Zu dem ».OB«-Pseudo gehört unmittelbar ein zweiter Pseudo ».EN«, der das mit »filename« gekennzeichnete File schließt. Dazu muß dieser Pseudo am Ende des Quelltextes stehen. Sollten mit dem ».AP« mehrere Quelltexte verkettet werden, muß der ».EN«-Pseudo am Schluß des letzten Quelltextes auftauchen.

Bei der Anwendung von Makros gab es auch einige Schwierigkeiten. Wird zum Beispiel von einem Makro (Ordnung 1) zweimal ein weiteres Makro (Ordnung 2) aufgerufen, meldet Hypra-Ass einen »label twice error«, vorausgesetzt, im Makro zweiter Ordnung befindet sich ein Label. Zum Beispiel würde folgendes Programm zu einer solchen Fehlermeldung führen:

```
10 -.BA $C000
20 -.MA MAK1
;MAKRODEFINITION 1. ORDNUNG
30 - ... MAK2
;MAKROAUFRUF 2. ORDNUNG
40 - ... MAK2
50 -.RT
60 -.MA MAK2
70 -LBL NOP
80 -.RT
90 -   ... MAK1
```

Dabei ist Mak1 das Makro 1. Ordnung und Mak2 das Makro 2. Ordnung. Alle Label in Makros zweiter oder dritter Ordnung sind untereinander global. Das heißt, daß in Makros zweiter Ordnung nur einmal das Label mit dem Namen »LBL« definiert werden dürfte.

Im Augenblick wird an einer Erweiterung gearbeitet, die diesen Mißstand beseitigt. Denn gerade beim intensiven Arbeiten mit Makros sind Makros zweiter und sogar dritter Ordnung unabdingbar. Ganz deutlich sieht man dies an dem Artikel »Wichtige Makros für Hypra-Ass« in dieser Ausgabe. Dort wurde ein Makro mit dem Namen »INCW (adresse)« definiert. Würde man die dort stehende 16-Bit-Addition ersetzen durch:

```
INC                 ADRESSE
BNE                 LBL
INC                 ADRESSE+1
LBL
```

könnte dieses Makro von keinem anderen Makro aus zweimal aufgerufen werden, weil durch das Label »LBL« ein »label twice error« erscheinen würde.

Der gleiche Fehler erscheint natürlich auch dann, wenn ein anderes Makro aufgerufen wird, das »LBL« als Label oder Variable benutzt. Denn die Ordnungszahl, die den beiden Labeln »LBL« zugewiesen wird, ist identisch.

Bedingte Assemblierung

Auch mit der bedingten Assemblierung wissen nur die wenigsten etwas anzufangen, obwohl sie gerade im Zusammenhang mit Makros eine große Rolle spielt. Dies soll an einem kleinen Beispiel demonstriert werden:

```
10  -.MA ADW (ADR1,ADR2,SUMME,RETTEN)
20  -.IF RETTEN !=! 1       ;NUR WENN RETTEN
                            = 1, WIRD
30  -           PHA         ;PHA IN DAS MASCHI-
                            NENPROGRAMM
40  -.EI
                            ;ASSEMBLIERT
50  -           CLC
60  -           LDA ADR1
70  -           ADC ADR2
80  -           STA SUMME
90  -           LDA ADR1+1
100 -           ADC ADR2+1
110 -           STA SUMME+1
120 -.IF RETTEN !=! 1       ;NUR WENN RETTEN
                            = 1, WIRD
130 -           PLA         ;PLA IN DAS
                            MASCHINENPROGRAMM
140 -.EI ;                  ASSEMBLIERT
150 -.RT
```

Dieses Makro addiert (ADR1,ADR1+1)+(ADR2, ADR2+1) und speichert das Ergebnis in den Speicherzellen (SUMME,SUMME+1). ADR1, ADR2 und SUMME können beliebige Speicherzellen oder Variablen sein. Soll der Inhalt des Akkumulators erhalten bleiben, wird für RETTEN eine 1, ansonsten eine beliebige andere Zahl eingegeben. Anhand des Übergabeparameters RETTEN erkennt der Assembler, ob das erzeugte Maschinenprogramm den Maschinenbefehl »PHA« beziehungsweise »PLA« enthalten soll oder nicht. Die Befehle zur bedingten Assemblierung zeigen also einzig und allein eine Wirkung beim Assemblieren. Durch sie wird bestimmt, welche Teile des Quelltextes im erzeugten Maschinenprogramm stehen und welche unter bestimmten Bedingungen übersprungen werden sollen. Schauen Sie sich nun noch einmal die Zeilen 20 und 120 an. Sie finden dort in der bedingten ».IF«-Abfrage den Operator »=«, der wie alle anderen Operatoren Ausrufezeichen erkennt Hypra-Ass, daß es sich bei dem Operator »=« um eine Rechenvorschrift aus dem Quelltext heraus handelt. Alle Operatoren können außer bei der bedingten As-

semblierung zum Beispiel auch bei der umnittelbaren Adressierung angewendet werden. Ein kleines Beispiel soll die Wirkung dieser Operatoren bei der unmittelbaren Adressierung verdeutlichen:

```
20 -.EQ VARIABLE1 = 10
30 -.EQ VARIABLE2 = 20
40 -          LDA #(VARIABLE !O! VARIABLE2)
```

Der Akkumulator wird mit einer Zahl geladen, die mit der Variablen »VARIABLE1« und »VARIABLE2« wie im Basic geORt wird. Das Ergebnis ist folglich 30.

Viele Maschinensprache-Anfänger verwechseln die Befehle zur bedingten Assemblierung mit normalen Basic-Befehlen. Deshalb möchte ich an einem kleinen Beispiel zeigen, was nicht mit der bedingten Assemblierung funktioniert:

```
10 -.BA $C000
20 -          INC $D020
30 -.GO 20
```

Was nicht funktioniert

Das Programm sollte die Bildschirmrahmenfarbe laufend um 1 inkrementieren. Wird der Assembler jedoch gestartet, ersetzt er den Befehl ».GO 20« nicht durch den Befehl »JMP adresse«. Vielmehr versucht er den gesamten Speicher von $C000 bis unendlich mit dem Befehl »INC $D020« zu füllen, denn es fehlt jegliche Abbruchbedingung. Nur mit einer Abbruchbedingung ist der ».GO«-Befehl sinnvoll. Sollte das Maschinenprogramm zehnmal hintereinander den Befehl »INC $D020« enthalten, könnte das so aussehen:

```
10 -.LI 1,3
20 -.BA $C000
30 -.EQ A = 0
40 -.EQ A = A + 1
50 -          INC $D020
60 -.IF A !<! 11
70 -.GO 40
80 -.EI
90 -          JMP $C000
```

Der Assembler überprüft in Zeile 60, ob die Variable »A« kleiner 11 ist. Trifft das zu, wird in Zeile 70 durch den ».GO«-Befehl zur Zeile 40 verzweigt und der Befehl »INC $D020« ein weiteres Mal assembliert. Sobald »A« gleich 10 ist, verzweigt der Assembler in die Zeile 90, übersetzt den Befehl »JMP $C000« und beendet den Assembliervorgang.

(Gerd Möllmann/ah/sk)

```
Name : hypra-ass              0801 1fd8
---------------------------------------
0801 : 60 08 00 00 9e 32 39 30  26
0809 : 31 3a 22 0d 91 05 0e 27  49
0811 : 68 59 50 52 41 2d 61 53  2e
0819 : 53 20 20 6d 41 4b 52 4f  88
0821 : 2d 61 53 53 45 4d 42 4c  9e
0829 : 45 52 0d 0d 20 28 63 29  9f
0831 : 20 31 39 38 35 20 62 59  d0
0839 : 20 20 67 45 52 44 20 6d  8e
0841 : 4f 45 4c 4c 4d 41 4e 4e  84
0849 : 0d 0d 20 28 63 4f 52 52  89
0851 : 45 43 54 45 44 20 76 45  9f
0859 : 52 53 49 4f 4e 29 00 00  bf
0861 : 00 20 73 00 20 6b 08 4c  64
0869 : ae a7 f0 0c e9 80 90 03  93
0871 : 4c f3 a7 68 68 4c 08 af  16
0879 : 60 0b 43 50 58 43 50 59  cd
0881 : 4c 44 58 4c 44 59 43 4d  46
0889 : 50 41 44 43 41 4e 44 44  13
0891 : 45 43 45 4f 52 49 4e 43  e2
0899 : 4c 44 41 41 53 4c 42 49  b3
08a1 : 54 4c 53 52 4f 52 41 52  6c
08a9 : 4f 4c 52 4f 52 53 42 43  ec
08b1 : 53 54 41 53 54 58 53 54  e7
08b9 : 59 4a 4d 50 4a 53 52 54  c6
08c1 : 58 41 54 41 58 54 59 41  07
08c9 : 54 41 59 54 53 58 54 58  99
08d1 : 53 50 48 50 50 4c 50 50  b2
08d9 : 48 41 50 4c 41 42 52 4b  65
08e1 : 52 54 49 52 54 53 4e 4f  b2
08e9 : 50 43 4c 43 53 45 43 43  49
08f1 : 4c 49 53 45 49 43 4c 56  ec
08f9 : 43 4c 44 53 45 44 44 45  f0
0901 : 59 49 4e 59 44 45 58 49  20
0909 : 4e 58 42 50 4c 42 4d 49  bd
0911 : 42 56 43 42 56 53 42 43  27
0919 : 43 42 43 53 42 4e 45 42  e9
0921 : 45 51 e4 c4 a6 a4 c5 65  52
0929 : 25 c6 45 e6 a5 06 24 46  87
0931 : 05 26 66 e5 85 86 84 4c  d7
0939 : 20 8a aa 98 a8 ba 9a 08  37
0941 : 28 48 68 00 40 60 ea 18  8a
0949 : 38 58 78 b8 d8 f8 88 c8  eb
0951 : ca e8 10 30 50 70 90 b0  c6
0959 : d0 f0 40 40 54 68 7b 7b  27
0961 : 7b 28 7b 28 7b a8 00 a8  23
0969 : 7b a8 a8 7b 3b 04 08 00  c6
0971 : c0 00 c0 d8 1f 02 02 02  8b
0979 : 02 03 03 02 01 0c fc 10  83
0981 : 10 14 18 04 04 2b 2d 2a  c5
0989 : 2f 5e 41 4f 3e 3d 3c 43  67
0991 : 41 4e 27 54 20 4e 55 4d  b2
0999 : 42 45 52 20 54 45 52 4d  6a
09a1 : 00 4e 4f 20 4d 4e 45 4d  97
09a9 : 4f 4e 49 43 00 49 4c 4c  ee
09b1 : 45 47 41 4c 20 52 45 47  ac
09b9 : 49 53 54 45 52 00 45 4e  40
09c1 : 44 20 4f 46 20 4c 49 4e  d8
09c9 : 45 20 45 58 53 50 45 43  ce
09d1 : 54 45 44 00 57 52 4f 4e  bb
09d9 : 47 20 41 44 44 52 45 53  9c
09e1 : 53 00 42 52 41 4e 43 48  33
09e9 : 20 54 4f 4f 20 46 41 52  cf
09f1 : 00 55 4e 4b 4e 4f 57 4e  f2
09f9 : 20 4c 41 42 45 4c 00 49  21
0a01 : 4c 4c 45 47 41 4c 20 4c  3d
0a09 : 41 42 45 4c 00 54 4f 4f  c5
0a11 : 20 4d 41 4e 59 20 4c 41  3c
0a19 : 42 45 4c 53 00 4e 4f 20  6b
0a21 : 4d 41 43 52 4f 20 54 4f  10
0a29 : 20 43 4c 4f 53 45 00 55  f2
0a31 : 4e 4b 4e 4f 57 4e 20 50  ab
0a39 : 53 45 55 44 4f 00 4c 41  b5
0a41 : 42 45 4c 20 44 45 43 4c  51
0a49 : 41 52 45 44 20 54 57 49  22
0a51 : 43 45 00 50 41 52 41 4d  87
0a59 : 45 54 45 52 00 52 45 54  b4
0a61 : 55 52 4e 00 90 09 a2 09  61
0a69 : ae 09 bf 09 d5 09 e3 09  f4
0a71 : f2 09 00 0a 0e 0a 1e 0a  e7
0a79 : 30 0a 3f 0a 54 0a 0a aa  d3
0a81 : bd 65 0a 85 22 bd 66 0a  e2
0a89 : 85 23 a9 0e 20 c3 ff 20  2c
0a91 : cc ff a9 00 85 13 20 d7  e8
0a99 : aa a0 00 b1 22 f0 06 20  cc
0aa1 : d2 ff c8 d0 f6 4c 62 a4  64
0aa9 : a9 00 85 0d 20 73 00 b0  54
0ab1 : 06 20 f3 bc 4c 0c 0b 20  ed
0ab9 : 13 b1 90 06 20 b1 0d 4c  e6
0ac1 : 0c 0b 20 79 00 10 03 4c  af
0ac9 : b1 ae c9 2b f0 de c9 24  1e
0ad1 : f0 20 c9 2d f0 16 c9 22  14
0ad9 : f0 15 c9 21 f0 17 c9 3e  56
0ae1 : f0 5a c9 3c f0 56 20 f1  1e
0ae9 : ae 4c 0c 0b 4c 0d af 4c  a6
0af1 : bd ae 4c 6a 0d 20 73 00  05
0af9 : c9 4e d0 0a 20 73 00 c9  90
0b01 : 21 d0 03 4c d0 ae a9 00  fe
0b09 : 4c 7f 0a 20 79 00 aa 10  fe
0b11 : 01 60 68 68 8a c9 21 d0  87
0b19 : 0c 20 73 00 aa 20 73 00  8c
0b21 : c9 21 d0 e2 8a a2 0a dd  ad
0b29 : 85 09 f0 05 ca d0 f8 f0  09
0b31 : 04 8a 18 69 a9 a2 00 86  6a
0b39 : 4d 4c bb ad 48 20 73 00  a4
0b41 : 20 f1 ae 20 f7 b7 aa 68  c2
0b49 : c9 3c f0 02 8a a8 20 a2  60
0b51 : b3 4c 0c 0b a9 9d a2 17  cf
0b59 : 8d 02 03 8e 03 03 a9 a9  bc
0b61 : a2 0a 8d 0a 03 8e 0b 03  84
0b69 : a9 47 a2 0f 8d 08 03 8e  83
0b71 : 09 03 a9 1f 85 2c a9 d8  5c
0b79 : 85 2b a9 00 8d d7 1f 20  53
0b81 : 44 a6 a2 fa 9a 4c 49 a8  a3
0b89 : a2 00 86 41 a0 00 b1 7a  bb
0b91 : dd 7b 08 d0 10 c8 b1 7a  4b
0b99 : dd 7c 08 d0 08 c8 b1 7a  53
0ba1 : dd 7d 08 f0 0e e6 41 e8  4c
0ba9 : e8 e8 e0 a8 d0 de a9 01  ff
0bb1 : 4c 7f 0a a6 41 bd 23 09  b5
0bb9 : 85 3b a9 02 4c fc a8 a9  29
0bc1 : 00 85 3e 20 73 00 c9 23  bc
0bc9 : f0 4f c9 28 d0 55 20 2a  65
0bd1 : 14 a5 3d d0 14 20 79 00  49
0bd9 : c9 2c f0 22 c9 29 d0 19  94
0be1 : 20 73 00 c9 2c f0 27 d0  7d
0be9 : 03 20 73 00 a5 41 c9 15  8f
0bf1 : d0 07 a9 09 85 3e 4c 6d  27
```

```
0bf9 : 0c a9 04 4c 7f 0a 20 73  14
0c01 : 00 a9 58 20 ff ae 20 f7  d6
0c09 : ae e6 3e 4c 6d 0c 20 73  e2
0c11 : 00 a9 59 20 ff ae 4c 6d  c2
0c19 : 0c 20 38 14 a9 06 85 3e  23
0c21 : 4c 6d 0c a9 07 85 3e 20  32
0c29 : 79 00 f0 3b c9 3b f0 37  ee
0c31 : a2 05 86 3e 20 2d 14 20  bb
0c39 : 79 00 c9 2c d0 22 20 73  30
0c41 : 00 aa 20 73 00 8a c9 58  39
0c49 : f0 0b c9 59 f0 05 a9 02  3e
0c51 : 4c 7f 0a c6 3e a5 3d d0  60
0c59 : 04 c6 3e c6 3e 4c 6d 0c  3d
0c61 : a9 0a 85 3e 4c 56 0c a9  33
0c69 : 07 85 3e 60 20 79 00 f0  7e
0c71 : fa c9 3b f0 f6 a9 03 4c  9e
0c79 : 7f 0a a6 3e e0 06 d0 0c  08
0c81 : a5 41 c9 04 b0 06 a5 3b  02
0c89 : e9 07 85 3b e0 09 f0 29  2b
0c91 : a5 41 c9 15 90 35 f0 14  8a
0c99 : c9 16 f0 10 c9 30 90 03  12
0ca1 : 4c 28 0d e0 07 d0 1f a9  27
0ca9 : 01 85 42 60 e0 08 f0 04  24
0cb1 : e0 0a d0 12 a9 03 85 42  5a
0cb9 : 60 a5 41 c9 15 d0 07 a9  bd
0cc1 : 6c 85 3b 4c b5 0c a9 04  b3
0cc9 : 4c 7f 0a a9 02 85 42 e0  a4
0cd1 : 08 f0 0d e0 0a d0 0a e6  ce
0cd9 : 42 a5 3b 18 69 08 85 3b  23
0ce1 : 60 a9 01 e0 00 f0 05 0a  22
0ce9 : ca 4c e4 0c a6 41 3d 5b  b4
0cf1 : 09 d0 11 a6 3e e0 02 f0  50
0cf9 : 04 e0 03 d0 c9 e8 e8 86  dd
0d01 : 3e 4c e2 0c a6 3e e0 04  87
0d09 : d0 0f a5 41 c9 02 d0 09  f4
0d11 : a9 03 85 42 a9 be 85 3b  03
0d19 : 60 bd 76 09 85 42 bd 7e  75
0d21 : 09 18 65 3b 85 3b 60 20  eb
0d29 : ad 0c c6 42 a5 fd c9 02  4c
0d31 : 90 f4 a5 3c 38 e5 fb aa  24
0d39 : a5 3d e5 fc a8 8a e9 02  20
0d41 : 85 3c b0 01 88 c0 00 f0  a1
0d49 : 10 c8 d0 12 c9 80 90 0e  33
0d51 : e6 3c c9 fe 90 02 c6 3c  54
0d59 : 60 c9 80 b0 01 60 a9 05  98
0d61 : 4c 7f 0a ed c9 80 90 e9  64
0d69 : 60 a2 0a a9 00 95 5d ca  8a
0d71 : d0 fb 20 73 00 90 0b 20  a6
0d79 : 13 b1 90 17 e9 07 c9 40  ea
0d81 : b0 11 e9 2f 20 7e bd a5  52
0d89 : 61 18 69 04 85 61 90 e2  3d
0d91 : 4c 7e b9 a5 61 e9 03 90  d2
0d99 : 05 85 61 4c 0c 0b 4c 08  9d
0da1 : af 85 62 84 63 a2 90 38  3a
0da9 : 4c 49 bc 00 00 01 00 00  d1
0db1 : a2 00 86 0d 86 0c 86 0e  96
0db9 : a5 7a a4 7b 85 49 84 4a  7d
0dc1 : 20 79 00 20 13 b1 b0 09  36
0dc9 : a9 20 85 81 a9 07 4c 7f  17
0dd1 : 0a 20 73 00 90 05 20 13  a0
0dd9 : b1 90 04 e8 4c d2 0d c9  14
0de1 : 27 f0 f8 e8 86 45 a9 36  81
0de9 : 85 01 a9 f9 a2 bf 86 60  9b
0df1 : 85 5f e4 30 90 4a d0 04  0c
0df9 : c5 2f 90 44 a0 01 b1 5f  9a
0e01 : c9 ff f0 0e cd ad 0d d0  e8
0e09 : 2a 88 b1 5f cd ac 0d d0  e8
0e11 : 22 c8 c8 b1 5f c5 45 d0  da
0e19 : 1a c8 b1 5f 85 47 c8 b1  09
0e21 : 5f 85 48 a0 00 b1 49 d1  bf
0e29 : 47 d0 08 c8 c4 45 d0 f5  99
0e31 : 4c a5 0e a5 5f 38 e9 07  f6
0e39 : a6 60 b0 b4 ca 4c ef 0d  bb
0e41 : 68 48 c9 bf d0 19 20 9f  cd
0e49 : 0e a6 fd f0 08 ca f0 08  f2
0e51 : a9 06 4c 7f 0a a9 00 2c  47
0e59 : a9 80 a0 00 4c a2 0d a5  c4
0e61 : 5f a4 60 c4 2e 90 06 d0  e4
0e69 : 07 c5 2d b0 03 4c b7 0e  42
0e71 : 85 2f 84 30 a0 00 ad ac  cf
0e79 : 0d 91 5f c8 ad ad 0d 91  df
0e81 : 5f c8 a5 45 91 5f c8 a5  d9
0e89 : 49 91 5f c8 a5 4a 91 5f  3d
0e91 : a5 5f a4 60 18 69 05 90  1d
0e99 : 01 c8 85 49 84 4a a9 37  39
0ea1 : 85 01 38 60 20 91 0e a9  db
0ea9 : 36 85 01 20 83 14 20 9f  7f
0eb1 : 0e 20 a6 0d 18 60 20 9f  5f
0eb9 : 0e a9 08 4c 7f 0a 00 00  70
0ec1 : db 23 a9 00 85 fd 85 fe  f5
0ec9 : 4c 62 08 a0 02 b1 7a f0  d6
0ed1 : 1f c8 b1 7a 85 39 c8 b1  b9
0ed9 : 7a 85 3a a4 7b a6 7a e8  e2
0ee1 : d0 01 c8 8e c1 0e 8c c2  7a
0ee9 : 0e a0 04 20 fb a8 18 60  73
0ef1 : 38 60 00 0f 1e 02 0f a5  b5
0ef9 : fd c9 02 90 43 ad 32 16  04
0f01 : 20 c3 ff a9 ff 85 3a a9  a0
0f09 : 00 8d ac 0d 8d ad 0d 8d  32
0f11 : af 0d a9 01 8d ae 0d ad  af
0f19 : e0 16 f0 12 20 d7 aa 20  2e
0f21 : 03 17 20 d7 aa 20 d7 aa  13
0f29 : ae db 16 20 c9 ff 20 3e  e8
0f31 : 15 20 a6 15 20 cc ff ad  66
0f39 : db 16 20 c3 ff 4c 7b e3  b8
0f41 : 20 78 14 4c e9 0f 24 9d  0f
0f49 : 10 03 4c c3 0e 20 1f 15  ef
0f51 : a2 fa 9a 20 8e a6 a5 2b  26
0f59 : a6 2c 8d 74 09 8e 75 09  f4
0f61 : a9 00 48 a2 c0 8d e0 16  99
0f69 : 8d 33 16 8d 72 09 8e 73  58
0f71 : 09 ad 72 09 ae 73 09 85  c4
0f79 : 2f 86 30 a9 00 8d c0 0e  b8
0f81 : 8d bf 0e 8d ac 0d 8d ad  e8
0f89 : 0d 8d af 0d a9 01 8d ae  21
0f91 : 0d 20 cc 0e b0 52 20 73  a8
0f99 : 00 c9 2e d0 42 20 73 00  16
0fa1 : c9 4d f0 1e c9 52 d0 37  f2
0fa9 : 20 73 00 c9 54 d0 10 ee  a6
0fb1 : c0 0e ad bf 0e cd c0 0e  4a
0fb9 : b0 25 a9 09 4c 7f 0a 4c  09
0fc1 : 08 af 20 73 00 a9 41 20  aa
0fc9 : ff ae 20 b1 0d a0 00 a5  7f
0fd1 : 7a 91 49 c8 a5 7b 91 49  8e
0fd9 : a9 ff 91 5f ee bf 0e 20  37
0fe1 : 09 a9 20 fb a8 4c 92 0f  9c
0fe9 : e6 fd a9 00 8d ac 0d 8d  c6
0ff1 : ad 0d 8d af 0d a9 01 8d  bb
0ff9 : ae 0d 20 8e a6 20 b4 16  72
1001 : 20 cc 0e 90 06 4c f8 0e  e0
1009 : 4c 96 10 a0 01 b1 7a c9  d4
1011 : 3b f0 f5 c9 20 f0 4e c9  d1
1019 : 2e f0 4a a5 fd c9 01 f0  1b
1021 : 0f c8 b1 7a f0 5b c9 20  a1
1029 : d0 f7 20 fb a8 4c 69 10  2f
1031 : a9 ff 85 81 20 73 00 20  49
1039 : b1 0d a9 20 85 81 b0 15  31
1041 : a5 4a cd 73 09 90 09 d0  c8
1049 : 0c a5 49 cd 72 09 b0 05  70
1051 : a9 0b 4c 7f 0a a5 fc a6  92
1059 : fb 20 91 14 20 79 00 c9  ad
1061 : 00 d0 08 f0 1c 20 73 00  7a
1069 : 20 79 00 c9 2e f0 55 20  7f
1071 : 5f 14 a0 00 b1 7a f0 14  dd
1079 : c9 3b f0 10 a9 03 4c 7f  01
1081 : 0a ad 33 16 f0 03 20 65  64
1089 : 17 20 73 00 20 09 a9 20  be
1091 : fb a8 4c fe 0f ad 33 16  2b
1099 : f0 ef ae 32 16 20 c9 ff  fc
10a1 : 20 57 17 ad c1 0e ae c2  b5
10a9 : 0e 85 5f 86 60 a2 00 20  7e
10b1 : 82 1b 20 d7 aa 20 cc ff  a3
10b9 : 20 73 00 20 09 a9 20 fb  ed
10c1 : a8 4c 01 10 a2 00 a0 01  80
10c9 : b1 7a dd 9c 14 d0 08 c8  3c
10d1 : b1 7a dd 9d 14 f0 0b e8  b1
10d9 : e8 e0 2c d0 e9 a9 0a 4c  03
10e1 : 7f 0a bd c9 14 48 bd c8  1a
10e9 : 14 48 ad 33 16 f0 03 20  28
10f1 : 65 17 a9 02 4c fc a8 20  1c
10f9 : 73 00 20 b1 0d a9 ff a0  0a
1101 : 01 91 5f 4c 0d 11 20 73  ed
1109 : 00 20 b1 0d a9 3d 20 ff  2c
1111 : ae a5 4a 48 a5 49 48 20  33
1119 : 2d 14 68 85 49 68 85 4a  9e
1121 : a5 3d a6 3c 20 91 14 4c  0d
1129 : 73 10 20 2a 14 8d 71 09  77
1131 : 8c 70 09 20 78 14 4c 73  7c
1139 : 10 a9 01 85 42 20 38 14  3d
1141 : 85 3b 20 d3 13 20 79 00  fe
1149 : c9 2c f0 f1 4c 73 10 a9  97
1151 : 01 85 42 a0 00 20 73 00  88
1159 : c9 22 f0 03 4c 99 ad c8  aa
1161 : b1 7a f0 11 c9 22 f0 0c  37
1169 : 85 3b 98 48 20 d3 13 68  79
1171 : a8 4c 60 11 c8 20 fb a8  48
1179 : 4c 73 10 a9 02 85 42 20  4e
1181 : 2a 14 85 3c 84 3b 20 d3  e8
1189 : 13 20 79 00 c9 2c f0 ef  ac
1191 : 4c 73 10 a6 fd ca f0 2a  be
1199 : 20 e7 ff 20 73 00 20 f4  52
11a1 : 14 a9 0e a0 6e a2 08 20  7e
11a9 : ba ff 20 c0 ff a2 0e 20  10
11b1 : c9 ff a5 fb 20 d2 ff a5  46
11b9 : fc 20 d2 ff 20 cc ff 4c  7b
11c1 : 8d 10 a9 80 85 fe 4c 8d  6d
11c9 : 10 a5 fd c9 02 90 0c a9  8d
11d1 : 00 85 fe a9 0e 20 c3 ff  7a
11d9 : 20 cc ff 4c 8d 10 20 09  d5
11e1 : a9 20 fb a8 a0 04 20 fb  51
11e9 : a8 20 73 00 c9 2e d0 ee  ad
11f1 : 20 73 00 c9 52 d0 e7 4c  e8
11f9 : 8d 10 ad ac 0d 0d ad 0d  99
1201 : 8d b0 0d 20 73 00 20 b1  49
1209 : 0d a5 7b 48 8d d2 13 a5  d8
1211 : 7a 48 8d d1 13 a9 36 48  35
1219 : 85 01 a0 01 b1 49 85 7b  da
1221 : 88 b1 49 85 7a a9 37 85  62
1229 : 01 20 79 00 c9 28 d0 4e  56
1231 : 20 bc 13 20 79 00 c9 28  87
1239 : d0 4a 20 2a 14 20 bc 13  d7
1241 : 20 8a 12 20 73 00 20 b1  4a
1249 : 0d a5 3d a6 3c 20 91 14  80
1251 : 20 79 00 c9 29 f0 14 c9  65
1259 : 2c d0 29 20 bc 13 20 79  14
1261 : 00 c9 2c d0 1f 20 ae 12  3d
1269 : 4c 3b 12 20 bc 13 20 79  b3
1271 : 00 c9 29 d0 0f 20 bc 13  c5
1279 : 20 a5 12 4c 8d 10 20 8a  69
1281 : 12 4c 79 12 a9 0c 4c 7f  85
1289 : 0a ad b0 0d d0 0d ad ae  c1
```

**Listing 1. »Hypra-Ass«.
Bitte mit dem MSE eingeben.**

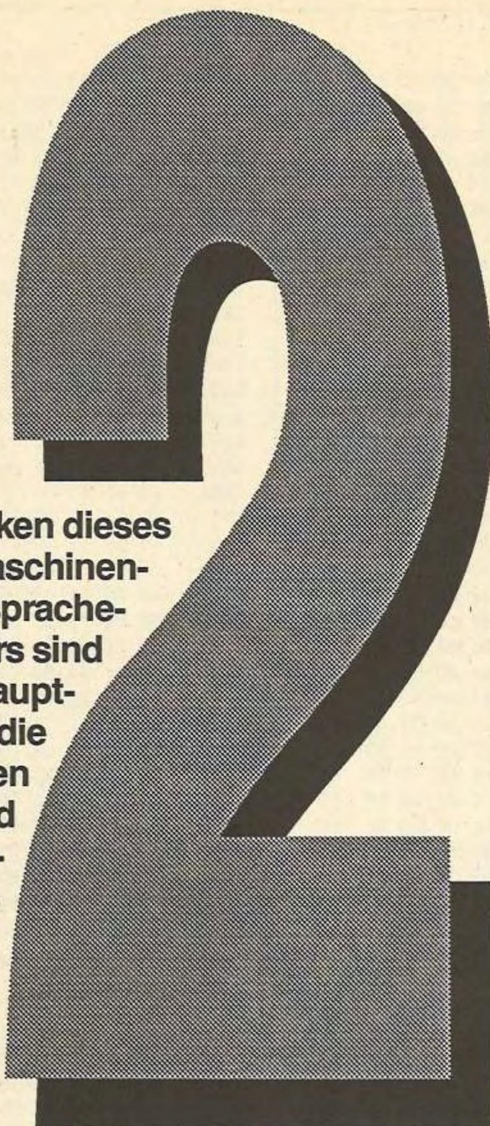
```
1291 : 0d ae af 0d 8d ac 0d 8e  12
1299 : ad 0d 60 ee ac 0d d0 03  3f
12a1 : ee ad 0d 60 ee ae 0d d0  ef
12a9 : 03 ee af 0d 60 ad b0 0d  01
12b1 : d0 09 a9 00 8d ac 0d 8d  fe
12b9 : ad 0d 60 ad ac 0d d0 03  37
12c1 : ce ad 0d ce ac 0d 60 68  08
12c9 : c9 36 d0 1d 68 85 7a 68  f3
12d1 : 85 7b 68 48 c9 36 d0 06  d5
12d9 : 20 bc 12 4c 8d 10 a9 00  65
12e1 : 8d ac 0d 8d ad 0d 4c 8d  49
12e9 : 10 a9 0d 4c 7f 0a a6 fd  7a
12f1 : ca d0 36 20 73 00 f0 14  d8
12f9 : c9 3b f0 10 20 b1 0d 20  a2
1301 : 79 00 c9 2c d0 06 20 73  17
1309 : 00 4c fd 12 a5 2f a6 30  c0
1311 : 8d 72 09 8e 73 09 20 09  fe
1319 : a9 c8 a6 7b 18 98 65 7a  10
1321 : 90 01 e8 8d 74 09 8e 75  d2
1329 : 09 4c 8d 10 a5 fd c9 02  33
1331 : 90 f7 20 cc ff a5 fb a6  c9
1339 : fc 8d 70 09 8e 71 09 20  12
1341 : 73 00 20 f4 14 a2 08 a0  13
1349 : 00 20 ba ff a9 00 85 0a  cd
1351 : ae 74 09 ac 75 09 20 75  1c
1359 : e1 a9 00 85 fd a5 fe f0  aa
1361 : 05 a2 0e 20 c9 ff 4c 72  f2
1369 : 0f 20 73 00 20 8a ad a5  be
1371 : 61 d0 0c 20 fd ae 20 2d  72
1379 : 14 20 a3 a8 4c fe 0f 4c  2d
1381 : 8d 10 20 2a 14 4c 7a 13  17
1389 : 20 73 00 20 8a ad a5 61  d6
1391 : f0 03 4c 73 10 20 09 a9  fe
1399 : c8 c8 c8 c8 20 fb a8 20  d5
13a1 : 73 00 c9 2e d0 ef 20 73  40
13a9 : 00 c9 45 d0 e8 20 73 00  57
13b1 : c9 4c f0 04 c9 49 d0 dd  43
13b9 : 4c 8d 10 a6 7a a4 7b ad  bb
13c1 : d1 13 85 7a ad d2 13 85  95
13c9 : 7b 8e d1 13 8c d2 13 60  ce
13d1 : 00 00 a6 fd ca f0 24 a5  4b
13d9 : fe d0 2c a5 fb a6 fc 85  f3
13e1 : 22 86 23 a6 42 a0 00 a5  58
13e9 : 3b 91 22 ca f0 0d c8 a5  b5
13f1 : 3c 91 22 ca f0 05 c8 a5  7d
13f9 : 3d 91 22 a5 42 18 65 fb  af
1401 : 85 fb 90 02 e6 fc 60 a2  05
1409 : 0e 20 c9 ff a6 42 a5 3b  23
1411 : 20 d2 ff ca f0 0d a5 3c  7a
1419 : 20 d2 ff ca f0 05 a5 3d  44
1421 : 4c 19 14 20 cc ff 4c fc  fb
1429 : 13 20 73 00 20 8a ad 20  77
1431 : f7 b7 85 3d 84 3c 60 20  f9
1439 : 73 00 c9 22 f0 09 20 2d  95
1441 : 14 c9 00 d0 16 98 60 a0  3d
1449 : 01 b1 7a 85 3c c8 b1 7a  38
1451 : c9 22 d0 07 c8 20 fb a8  0f
1459 : a5 3c 60 4c 48 b2 20 89  6c
1461 : 0b 20 c0 0b 20 7b 0c a6  69
1469 : fd ca f0 08 ad 33 16 f0  b7
1471 : 03 20 4e 16 4c d3 13 ad  e6
1479 : 70 09 ae 71 09 85 fb 86  01
1481 : fc 60 a0 01 b1 49 85 62  36
1489 : 88 b1 49 85 63 84 70 60  ca
1491 : a0 01 91 49 88 8a 91 49  f5
1499 : 84 70 60 45 51 42 41 42  c7
14a1 : 59 54 58 57 4f 4f 42 45  28
14a9 : 4e 4d 41 2e 2e 52 54 43  01
14b1 : 4f 41 50 4f 4e 47 4f 49  8e
14b9 : 46 45 4c 45 49 47 4c 4c  f6
14c1 : 49 53 59 53 54 44 50 06  29
14c9 : 11 2a 11 39 11 4f 11 7b  22
14d1 : 11 93 11 c9 11 de 11 fa  6c
14d9 : 11 c7 12 ee 12 2c 13 69  d2
14e1 : 13 82 13 88 13 95 13 b8  a7
14e9 : 13 f7 10 f2 15 bc 16 3c  62
14f1 : 17 78 17 a6 7a a4 7b e8  6b
14f9 : d0 01 c8 86 bb 84 bc a0  61
1501 : 00 c8 b1 7a c9 22 d0 f9  06
1509 : 88 84 b7 60 45 4e 44 20  e5
1511 : 4f 46 20 41 53 53 45 4d  33
1519 : 42 4c 59 20 20 00 ad 0e  b1
1521 : dc 09 80 8d 0e dc ad 0f  f0
1529 : dc 29 fe 8d 0f dc a9 00  8a
1531 : 8d 0b dc 8d 0a dc 8d 09  fc
1539 : dc 8d 08 dc 60 a9 00 8d  e8
1541 : 0b dc 20 d7 aa 20 57 17  f5
1549 : a9 0d a0 15 20 1e ab ad  41
1551 : 0a dc 29 0f 18 69 30 20  c3
1559 : d2 ff a9 3a 20 d2 ff ad  d0
1561 : 09 dc aa 29 f0 4a 4a 4a  c7
1569 : 4a 18 69 30 20 d2 ff 8a  cd
1571 : 29 0f 18 69 30 20 d2 ff  a4
1579 : a9 2e 20 d2 ff ad 08 dc  e3
1581 : 29 0f 18 69 30 20 d2 ff  b4
1589 : 4c d7 aa 42 41 53 45 20  b8
1591 : 3d 20 24 00 20 20 4c 41  9e
1599 : 53 54 20 42 59 54 45 20  f4
15a1 : 41 54 20 24 00 20 57 17  26
15a9 : a9 8c a0 15 20 1e ab ad  60
15b1 : 71 09 20 37 16 ad 70 09  38
15b9 : 20 37 16 a9 95 a0 15 20  23
15c1 : 1e ab a4 fc a6 fb d0 01  0d
15c9 : 88 ca 98 20 37 16 8a 20  6f
15d1 : 37 16 4c d7 aa 48 59 50  14
15d9 : 52 41 2d 41 53 53 20 20  d0
15e1 : 41 53 53 45 4d 42 4c 45  ec
15e9 : 52 4c 49 53 54 49 4e 47  75
15f1 : 3a 00 a6 fd ca f0 37 20  e6
15f9 : 73 00 20 19 e2 a5 b8 8d  f1
1601 : 32 16 20 c0 ff b0 2c a9  e8
1609 : ff 8d 33 16 ae 32 16 20  74
1611 : c9 ff 20 d7 aa 20 d7 aa  3d
1619 : 20 57 17 a9 d6 a0 15 20  e7
1621 : 1e ab 20 d7 aa 20 d7 aa  78
1629 : 20 cc ff 20 65 17 4c 8d  0f
1631 : 10 00 00 4c f9 e0 48 4a  27
1639 : 4a 4a 4a 20 42 16 68 29  08
1641 : 0f 18 69 30 c9 3a 90 02  71
1649 : 69 06 4c d2 ff ae 32 16  8d
1651 : 20 c9 ff a2 00 ec f6 0e  09
1659 : b0 08 a9 20 20 d2 ff e8  e6
1661 : d0 f3 a5 fc 20 37 16 a5  93
1669 : fb 20 37 16 20 52 17 a6  43
1671 : 42 a5 3b 20 37 16 ca f0  8a
1679 : 10 a5 3c 20 37 16 ca f0  a0
1681 : 0b a5 3d 20 37 16 4c 90  29
1689 : 16 20 4d 17 20 4d 17 20  ef
1691 : 52 17 a9 3a 20 d2 ff 4c  52
1699 : cc ff ad c1 0e ae c2 0e  86
16a1 : 85 5f 86 60 ae 32 16 20  99
16a9 : c9 ff 20 4d 1b 20 d7 aa  8b
16b1 : 4c cc ff ad 33 16 f0 03  c7
16b9 : 4c 9b 16 60 a6 fd ca f0  cc
16c1 : 15 20 73 00 20 19 e2 a2  5f
16c9 : 05 b5 b7 9d da 16 ca 10  f4
16d1 : f8 a9 ff 8d e0 16 4c 8d  5a
16d9 : 10 00 00 00 00 00 00 00  ea
16e1 : 4c f9 e0 53 59 4d 42 4f  74
16e9 : 4c 53 20 49 4e 20 41 4c  94
16f1 : 50 48 41 42 45 54 49 43  a1
16f9 : 41 4c 20 4f 52 44 45 52  53
1701 : 3a 00 a2 05 bd da 16 95  bb
1709 : b7 ca 10 f8 20 c0 ff b0  b2
1711 : cf ae db 16 20 c9 ff 20  82
1719 : d7 aa 20 d7 aa 20 d7 aa  a9
1721 : 20 57 17 a9 e4 a0 16 20  d4
1729 : 1e ab 20 d7 aa 20 d7 aa  80
1731 : 20 7a 1e 20 90 1d 20 d7  3c
1739 : aa 4c cc ff a6 fd ca f0  a4
1741 : 08 a9 0e 20 c3 ff 4c f8  05
1749 : 0e 4c 8d 10 a9 20 20 d2  a4
1751 : ff a9 20 4c d2 ff a2 00  6e
1759 : ec f6 0e b0 06 20 52 17  33
1761 : e8 d0 f5 60 ae 32 16 20  50
1769 : c9 ff 20 57 17 a2 0d 20  20
1771 : 52 17 ca d0 fa 4c cc ff  61
1779 : 20 38 14 8d f4 0e 20 fd  a8
1781 : ae 20 3f 14 8d f5 0e 20  93
1789 : fd ae 20 3f 14 8d f6 0e  73
1791 : 20 fd ae 20 3f 14 8d f7  1a
1799 : 0e 4c 8d 10 a9 00 85 fd  df
17a1 : 85 fc a6 02 f0 0c 30 42  43
17a9 : ca ca f0 2d ca f0 2d ca  39
17b1 : f0 30 20 60 a5 86 7a 84  4f
17b9 : 7b 20 73 00 aa f0 f3 c9  b7
17c1 : 5f f0 13 c9 2f f0 18 a2  b6
17c9 : ff 86 3a 20 79 00 90 2c  d0
17d1 : 20 79 a5 4c e1 a7 4c 1e  69
17d9 : 1f 4c 9c 1a 4c 81 18 4c  53
17e1 : 5c 1e 4c 48 1c a9 00 85  83
17e9 : 02 60 a5 02 29 7f 85 02  6e
17f1 : 20 6b 18 20 4d 1b 20 d7  af
17f9 : aa 4c 9d 17 20 6b a9 a6  65
1801 : 7a e8 a0 05 bd 00 02 99  cf
1809 : fb 01 f0 49 c9 3b f0 27  73
1811 : c9 2e f0 23 c9 20 f0 05  fd
1819 : e8 c8 4c 05 18 e8 bd 00  d9
1821 : 02 f0 31 c9 20 f0 f6 c9  1a
1829 : 22 f0 18 c9 3b f0 07 c8  eb
1831 : 99 fb 01 4c 1e 18 c8 99  8b
1839 : fb 01 e8 bd 00 02 d0 f6  e8
1841 : 4c 55 18 c8 99 fb 01 e8  a6
1849 : bd 00 02 f0 07 c9 22 d0  8e
1851 : f2 4c 30 18 c8 99 fd 01  cc
1859 : 99 fb 01 c6 7b a9 ff 85  19
1861 : 7a a5 02 09 80 85 02 4c  24
1869 : a2 a4 a9 91 20 d2 ff a2  d8
1871 : 00 a9 20 20 d2 ff e8 e0  e4
1879 : 27 d0 f6 a9 00 85 d3 60  37
1881 : ad 3d 03 ae 3c 03 20 cd  5b
1889 : bd a2 04 20 2a 1f a9 2d  39
1891 : 20 d2 ff ad 3d 03 85 15  fc
1899 : ad 3c 03 85 14 18 6d 40  0e
18a1 : 03 8d 3c 03 90 03 ee 3d  32
18a9 : 03 20 60 a5 86 7a c8 84  f1
18b1 : 7b a2 00 ad 00 02 d0 06  92
18b9 : 20 e6 17 4c 9d 17 4c 03  65
18c1 : 18 e6 7a 20 eb b7 a5 14  2a
18c9 : a4 15 8d 3c 03 8c 3d 03  72
18d1 : 8e 40 03 a9 03 85 02 4c  72
18d9 : 9d 17 20 73 00 90 09 f0  03
18e1 : 07 c9 2d f0 03 4c 08 af  48
18e9 : 20 6b a9 20 13 a6 20 79  07
18f1 : 00 f0 0c c9 2d d0 ee 20  fb
18f9 : 73 00 20 6b a9 d0 e6 a5  ea
1901 : 14 05 15 d0 06 a9 ff 85  b0
1909 : 14 85 15 60 a0 01 b1 5f  c9
1911 : f0 13 c8 b1 5f aa c8 b1  c5
1919 : 5f c5 15 d0 04 e4 14 f0  54
1921 : 02 b0 03 18 24 38 60 a0  06
1929 : 00 b1 5f aa c8 b1 5f 85  d2
1931 : 60 86 5f 60 4c 48 b2 20  ca
1939 : db 18 a5 5f a6 60 85 19  2b
1941 : 86 1a a5 14 25 15 aa e8  38
1949 : f0 06 e6 14 d0 02 e6 15  5b
```

```
1951 : 20 13 a6 a5 5f a6 60 85  11
1959 : 24 86 25 38 e5 19 8a e5  2e
1961 : 1a 90 d1 a5 2d e5 24 85  8a
1969 : 5f a5 2e e5 25 85 60 18  13
1971 : a5 19 65 5f 85 2d a5 1a  75
1979 : 65 60 85 2e a0 00 b1 24  4f
1981 : 91 19 e6 19 d0 02 e6 1a  68
1989 : e6 24 d0 02 e6 25 a5 5f  e3
1991 : 38 e9 01 85 5f a5 60 e9  27
1999 : 00 85 60 10 e1 20 59 a6  48
19a1 : 20 33 a5 4c 74 a4 20 db  f2
19a9 : 18 20 2c a8 20 0d 19 b0  22
19b1 : 0c 20 4d 1b 20 d7 aa 20  30
19b9 : 28 19 4c aa 19 4c 74 a4  e5
19c1 : 20 9b b7 86 fe 20 fd ae  b4
19c9 : 20 8a ad 20 f7 b7 a6 fe  73
19d1 : 9d 61 03 98 9d 43 03 20  33
19d9 : fd ae 20 8a ad 20 f7 b7  b2
19e1 : a6 fe 9d 9d 03 98 9d 7f  8c
19e9 : 03 4c 74 a4 bd 43 03 85  d1
19f1 : 14 bd 61 03 85 15 86 fe  b6
19f9 : 20 13 a6 a6 fe bd 7f 03  03
1a01 : 85 14 bd 9d 03 85 15 60  25
1a09 : 20 9e b7 20 ed 19 20 44  1b
1a11 : e5 4c aa 19 20 9b b7 86  b5
1a19 : fe 20 fd ae 20 eb b7 8e  da
1a21 : 40 03 a5 14 a6 15 8d 3c  90
1a29 : 03 8e 3d 03 a6 fe 20 ed  e2
1a31 : 19 ad 3c 03 9d 43 03 ad  ec
1a39 : 3d 03 9d 61 03 20 0d 19  23
1a41 : b0 1d ad 3d 03 91 5f 88  de
1a49 : ad 3c 03 91 5f 18 6d 40  f4
1a51 : 03 8d 3c 03 90 03 ee 3d  e2
1a59 : 03 20 28 19 4c 3e 1a ad  14
1a61 : 3c 03 38 ed 40 03 a6 fe  9f
1a69 : 9d 7f 03 b0 03 ce 3d 03  3e
1a71 : ad 3d 03 9d 9d 03 4c 74  3d
1a79 : a4 a9 02 85 02 a6 2c a5  74
1a81 : 2b 8e 3d 03 8d 3c 03 a9  bd
1a89 : ff 85 14 85 15 20 13 a6  ed
1a91 : a5 5f a6 60 85 2b 86 2c  c0
1a99 : 4c c6 1a ae 3d 03 ad 3c  c0
1aa1 : 03 86 2c 85 2b 20 e6 17  21
1aa9 : 4c 74 a4 20 9b b7 8e 49  a1
1ab1 : 1f 4c 9d 17 20 73 00 20  1e
1ab9 : 57 e2 a0 00 ae 49 1f 4c  f4
1ac1 : ba ff a9 01 2c a9 00 85  20
1ac9 : 0a 20 b5 1a 20 6f e1 4c  32
1ad1 : 74 a4 20 b5 1a 20 59 e1  22
1ad9 : 4c 74 a4 20 9b b7 20 ed  60
1ae1 : 19 20 83 1c 20 fd ae 20  5c
1ae9 : ad 1c 20 0d 19 90 06 20  bd
1af1 : e6 17 4c 74 a4 a6 60 a5  51
1af9 : 5f 18 69 04 85 5d 90 01  c7
1b01 : e8 86 5e a9 03 85 45 a0  ac
1b09 : ff e6 45 c8 b1 5d f0 22  f4
1b11 : c4 ba f0 22 d1 bb f0 f3  59
1b19 : b1 bb c9 3f f0 ed c8 b1  07
1b21 : 5d d0 fb c4 ba f0 1e 90  4a
1b29 : 1c e6 5d d0 da e6 5e 4c  21
1b31 : 08 1b c4 ba d0 0f a5 02  6f
1b39 : c9 04 d0 03 4c 04 1c 20  2e
1b41 : 4d 1b 20 d7 aa 20 28 19  9d
1b49 : 4c eb 1a 03 a0 02 b1 5f  11
1b51 : aa c8 b1 5f 85 62 86 63  04
1b59 : a2 90 38 20 49 bc 20 df  10
1b61 : bd 20 87 b4 20 a6 b6 8d  d4
1b69 : 4c 1b 20 24 ab ae 4c 1b  67
1b71 : e0 05 b0 08 a9 20 20 d2  c3
1b79 : ff e8 d0 f4 a9 2d 20 d2  e9
1b81 : ff a0 04 b1 5f c9 3b f0  1b
1b89 : 57 c9 2e f0 53 b1 5f f0  91
```

```
1b91 : 5b c9 20 f0 08 20 d2 ff  c4
1b99 : e8 c8 4c 8e 1b 20 d2 ff  c8
1ba1 : c8 e8 ec f4 0e b0 08 a9  91
1ba9 : 20 20 d2 ff 4c a2 1b a9  27
1bb1 : 03 85 22 b1 5f f0 35 20  c8
1bb9 : d2 ff c8 e8 c6 22 d0 f3  83
1bc1 : 20 3f ab e8 b1 5f f0 24  ab
1bc9 : c9 3b f0 07 20 d2 ff e8  b7
1bd1 : c8 d0 f1 ec f5 0e b0 08  be
1bd9 : a9 20 20 d2 ff e8 d0 f3  67
1be1 : b1 5f f0 08 20 d2 ff e8  e9
1be9 : c8 4c e1 1b 60 20 9b b7  98
1bf1 : 20 ed 19 20 83 1c 20 fd  e8
1bf9 : ae 20 95 1c a9 04 85 02  75
1c01 : 4c e5 1a a5 14 48 a5 15  bf
1c09 : 48 a2 05 a0 02 b1 5f 85  2e
1c11 : 14 c8 b1 5f 85 15 c8 c4  8f
1c19 : 45 f0 09 b1 5f 9d fb 01  24
1c21 : e8 4c 17 1c a0 00 c4 b7  05
1c29 : f0 0a b1 b8 9d fb 01 e8  31
1c31 : c8 4c 27 1c a4 ba b1 5d  0e
1c39 : 9d fb 01 f0 05 e8 c8 4c  86
1c41 : 37 1c 8a a8 4c a2 a4 20  eb
1c49 : 13 a6 20 4d 1b 20 d7 aa  c9
1c51 : 68 85 15 68 85 14 a6 fe  60
1c59 : bd 7f 03 85 14 bd 9d 03  f3
1c61 : 85 15 a5 45 18 65 b7 a8  60
1c69 : b1 5f f0 13 a6 60 98 18  68
1c71 : 65 5f 85 5d 90 01 e8 86  55
1c79 : 5e 88 84 45 4c 08 1b 4c  ef
1c81 : 46 1b a9 03 85 b9 85 bc  d5
1c89 : a9 b0 85 b8 a9 d8 85 bb  f2
1c91 : 60 4c 48 b2 20 9e ad 20  6e
1c99 : 82 b7 c9 26 b0 f3 85 b7  5e
1ca1 : a0 00 b1 22 91 b8 c8 c4  7d
1ca9 : b7 d0 f7 60 20 9e ad 20  c0
1cb1 : 82 b7 f0 dd c9 26 b0 d9  4b
1cb9 : 85 ba a0 00 b1 22 91 bb  ad
1cc1 : c8 c4 ba d0 f7 60 a9 08  ed
1cc9 : 85 ba 20 b4 ff a9 6f 85  60
1cd1 : b9 20 96 ff 20 a5 ff 20  af
1cd9 : d2 ff c9 0d d0 f6 20 ab  5b
1ce1 : ff 4c 74 a4 a9 24 8d 00  aa
1ce9 : 01 20 d7 aa a9 01 a8 a2  d0
1cf1 : 00 20 bd ff a2 08 a0 60  1e
1cf9 : 20 ba ff 20 d5 f3 a5 ba  83
1d01 : 20 b4 ff a5 b9 20 96 ff  27
1d09 : a9 00 85 90 a0 03 84 fb  52
1d11 : 20 a5 ff 85 fe a4 90 d0  ad
1d19 : 30 20 a5 ff a4 90 d0 29  27
1d21 : a4 fb 88 d0 e9 a6 fe 20  0f
1d29 : cd bd a9 20 20 d2 ff 20  1c
1d31 : a5 ff a6 90 d0 13 aa f0  c4
1d39 : 06 20 d2 ff 4c 30 1d a9  12
1d41 : 0d 20 d2 ff a0 02 4c 0f  7c
1d49 : 1d 20 42 f6 20 e6 17 4c  14
1d51 : c7 1c a9 08 a0 01 85 ba  2f
1d59 : 20 b1 ff a9 6f 85 b9 20  d1
1d61 : 93 ff b1 7a f0 07 20 a8  c9
1d69 : ff c8 4c 63 1d 20 ae ff  d9
1d71 : 4c 74 a4 a5 30 c5 2e d0  61
1d79 : 06 a5 2f c5 2d f0 0d 20  a5
1d81 : 73 00 c9 21 d0 03 20 7a  25
1d89 : 1e 20 90 1d 4c 74 a4 a2  bf
1d91 : bf a9 f9 85 45 86 46 e4  bf
1d99 : 30 90 7d d0 04 c5 2f 90  d7
1da1 : 77 20 2c a8 a9 36 85 01  ad
1da9 : a0 00 b1 45 c8 11 45 d0  2a
1db1 : 56 a0 06 b1 45 85 62 88  2a
1db9 : b1 45 85 63 88 b1 45 85  11
1dc1 : 23 88 b1 45 85 22 88 b1  2c
1dc9 : 45 85 47 a9 37 85 01 20  bc
```

```
1dd1 : d7 aa 20 57 17 a0 00 a2  ac
1dd9 : 00 b1 22 20 d2 ff c8 e8  61
1de1 : c4 47 d0 f5 e8 20 3f ab  1f
1de9 : ec f7 0e 90 f7 20 3f ab  3b
1df1 : a9 3d 20 d2 ff 20 3f ab  f1
1df9 : a9 24 20 d2 ff a5 62 20  0e
1e01 : 37 16 a5 63 20 37 16 a5  78
1e09 : 45 a6 46 38 e9 07 b0 01  d6
1e11 : ca a0 37 84 01 4c 94 1d  89
1e19 : 60 4d 56 4c 53 41 44 4e  2c
1e21 : 45 47 54 50 46 58 52 49  2c
1e29 : 4b 40 21 4f 42 55 43 1a  d7
1e31 : 1a 1a 1a 18 19 1a 19 1a  dd
1e39 : 1f 19 1a fc 1b 1c 1c 1d  48
1e41 : 1d 1f 1f 1f 1f 79 c2 c5  ee
1e49 : d2 c1 37 14 a6 ab 92 c0  e0
1e51 : db e1 ed e4 c6 52 73 2e  5e
1e59 : 49 81 b9 20 73 00 b0 03  d5
1e61 : 4c 09 1a a2 17 dd 19 1e  0e
1e69 : f0 06 ca d0 f8 4c 08 af  9a
1e71 : bd 2f 1e 48 bd 45 1e 48  66
1e79 : 60 a9 36 85 01 a5 30 48  7b
1e81 : a5 2f 48 20 e6 17 4c a4  76
1e89 : 1e a5 02 f0 0b a5 19 a6  a8
1e91 : 1a 85 2f 86 30 4c 84 1e  be
1e99 : a9 37 85 01 68 85 2f 68  a0
1ea1 : 85 30 60 a9 f9 a2 bf 85  4a
1ea9 : 19 86 1a 38 e9 07 85 1b  b6
1eb1 : b0 01 ca 86 1c e4 30 d0  b1
1eb9 : 06 90 ce c5 2f 90 ca a0  57
1ec1 : 02 b1 19 85 21 b1 1b 85  aa
1ec9 : 22 c8 b1 19 85 1d b1 1b  1d
1ed1 : 85 1f c8 b1 19 85 1e b1  e8
1ed9 : 1b 85 20 a0 00 b1 1d d1  79
1ee1 : 1f f0 04 b0 1a 90 11 c8  8b
1ee9 : c4 21 f0 04 c4 22 d0 ed  77
1ef1 : a5 21 c5 22 f0 02 b0 07  cc
1ef9 : a5 1b a6 1c 4c a8 1e a0  1d
1f01 : 06 b1 19 48 88 10 fa a0  65
1f09 : 06 b1 1b 91 19 88 10 f9  eb
1f11 : c8 68 91 1b c0 06 90 f8  46
1f19 : 84 02 4c f9 1e 20 73 00  a1
1f21 : 20 a0 aa 4c 74 a4 20 3f  31
1f29 : ab e4 d3 b0 f9 60 a9 ff  9a
1f31 : a0 01 91 2b 20 33 a5 a5  99
1f39 : 22 a6 23 38 69 01 90 01  61
1f41 : e8 85 2d 86 2e 4c 74 a4  68
1f49 : 08 a9 1f a2 d8 20 cd bd  83
1f51 : 20 3f ab a5 2c a6 2b 20  95
1f59 : cd bd 20 3f ab a5 2e a6  e3
1f61 : 2d 20 cd bd 20 3f ab a9  c7
1f69 : 00 38 e5 2d 85 63 a9 a0  00
1f71 : e5 2e 85 62 20 d1 bd a9  f6
1f79 : 66 a0 e4 20 1e ab 4c 74  c6
1f81 : a4 20 2a 14 a5 14 85 2b  aa
1f89 : a5 15 85 2c 20 44 a6 4c  f7
1f91 : 74 a4 20 9b b7 c9 2c d0  ef
1f99 : 16 e0 04 b0 12 86 21 20  51
1fa1 : 9b b7 a4 21 c0 02 f0 0a  59
1fa9 : 8a 99 f4 0e 4c 74 a4 4c  92
1fb1 : 48 b2 8a 8d f6 0e 4c 74  a1
1fb9 : a4 a9 00 85 21 20 73 00  c3
1fc1 : c9 52 f0 06 c9 48 d0 e7  a2
1fc9 : e6 21 20 9b b7 8a a4 21  60
1fd1 : 99 20 d0 4c 74 a4 00 00  a4
```

Listing 1. »Hypra-Ass« (Schluß)




Die Stärken dieses Super-Maschinensprache-Monitors sind hauptsächlich die mächtigen Such- und Trace-Befehle zum Austesten von Programmen in Maschinensprache. Der SMON enthält auch einen vollständigen Diskmonitor und einen Disassembler, der auch illegale Opcodes disassembliert. Ein Programm, mit dem auch Profis gern arbeiten.

Ich kann mich noch gut an unsere ersten Schritte in Maschinensprache erinnern. Ausgerüstet mit einer Befehlsliste für den 6502-kompatiblen Prozessor des C 64 und einem in Basic geschriebenen »Mini-Monitor« entstanden Programme, die 3 und 5 addieren und das Ergebnis im Speicher ablegen konnten. Dazu mußten wir die Befehlscodes aus der Liste heraussuchen und dann in den Speicher »POKEn«. Jeder Sprung mußte von Hand ausgerechnet werden, jeder falsch herausgesuchte Befehl führte zum Programmabsturz. Der erste Disassembler - ein Programm zur Anzeige der Maschinenbefehle in Assemblersprache - war für uns die Offenbarung. Von nun an konnten wir Maschinenprogramme analysieren und daraus lernen. Zum Verständnis der Maschinensprache ist es nämlich noch weit mehr als bei anderen Sprachen wichtig, vorhandene Programme zu verstehen und sich dabei die wichtigsten Techniken anzueignen.

Mit der Zeit wuchsen unsere Ansprüche, ein Assembler mußte her, um die neugewonnenen Erkenntnisse auch auszuprobieren. Das war zuerst wieder ein Basic-Programm, langsam und wenig komfortabel, aber immerhin. Wir schrieben unsere ersten kleinen Routinen, vor allem, um vorhandene Maschinenprogramme unseren eigenen Wünschen anzupassen. So entstand im Laufe eines Jahres SMON. Immer neue Befehle und Routinen kamen hinzu, bis wir endlich zufrieden waren.

Was bietet SMON?

Zunächst ist alles enthalten, was zum »Standard« gehört: Memory-Dump, also die Anzeige des Speicherinhalts in Hex-Bytes, mit Änderungsmöglichkeiten, ein Disassembler mit Änderungsmöglichkeit sowie Routinen zum Laden, Speichern und Starten von Maschinenprogrammen. Darüber hinaus gibt es einen kleinen Direktassembler, der sogar Labels verarbeitet, Befehle zum Verschieben im Speicher mit und ohne Umrechnen der Adressen und Routinen zum Umrechnen von Hex-, Dezimal- und Binärzahlen. Der besondere Clou von SMON liegt aber zweifellos in seinen leistungsfähigen Suchroutinen und vor allem im Trace-Modus. Damit lassen sich Maschinenprogramme Schritt für Schritt abarbeiten und kontrollieren.

Der Monitor benötigt für alle Eingaben die hexadezimale Schreibweise, das heißt zu den Zahlen 1 bis 9 kommen noch die Buchstaben A (für dez. 10) bis F (für dez. 15) hinzu.

Bei der Eingabe von Adressen ist folgendes zu beachten: [ANFADR] bedeutet exakt die Startadresse, [ENDADR] bedeutet hierbei die erste Adresse hinter dem gewählten Bereich. Im Normalfall ist die Eingabe mit und ohne Leerzeichen zulässig. Beim Abweichen von dieser Regel wird darauf besonders verwiesen. Tippen Sie zuerst das Hauptprogramm (Listing 1) mit dem MSE ab. Befindet sich SMON auf Ihrer Diskette, kann er mit LOAD "SMON $C000",8,1 geladen und mit dem Befehl SYS 49152 gestartet werden. Geben Sie vor dem SYS-Befehl aber NEW ein, um einen späteren »OUT OF MEMORY« zu verhindern.

Assemblieren

Syntax: A [ANFADR]

Assemblierung beginnt bei der angegebenen Adresse

Beispiel:

A 4000 Beginn bei Startadresse $4000

Nach Eingabe von »RETURN« erscheint auf dem Bildschirm die gewählte Adresse mit einem blinkenden Cursor. Die Befehle werden so eingegeben, wie sie der Disassembler zeigt: LDY #00 oder LDA 400E,Y und so weiter. »RETURN« schließt die Eingabe der Zeile ab.

Bei einer fehlerhaften Eingabe springt der Cursor wieder in die Anfangsposition zurück. Ansonsten wird der Befehl disassembliert und nach Ausgabe der Hex-Bytes gelistet. Zur Korrektur vorhergehender Zeilen gehen Sie mit dem Cursor zur Anfangsposition (hinter die Adresse) zurück, schreiben den Befehl neu und gehen nach »RETURN« mit dem Cursor wieder in die letzte Zeile. Falls Ihnen bei Sprüngen (Branch-Befehl, JSR und JMP) die Zieladressen noch nicht bekannt sind, geben Sie einfach sogenannte »Label« ein.

N der Profi-Monitor

Ein Label besteht aus dem Buchstaben »M« (für Marke) und einer zweistelligen Hex-Zahl von 01 bis 30.

Beispiel: BCC M01

Wenn Sie die Zieladresse für diesen Sprung erreicht ha-

Mit Bytes spielen

ben, dann kennzeichnen Sie diese mit eben dieser »Marke«.

Beispiel: M01 LDY #00

Einzelne Bytes nimmt der Assembler an, indem Sie diese mit einem Punkt kennzeichnen: beispielsweise .00 oder .AB. In diesem Modus werden die Eingaben natürlich nicht disassembliert.

Nach Beendigung des Assemblierens geben Sie »F« ein. Danach sehen Sie alle Ihre Eingaben noch einmal aufgelistet und korrigieren dann bei Bedarf wie beim Disassembler (!) angegeben.

Probieren Sie einmal das folgende Beispiel:

A 4000

Der Assembler meldet sich mit: »4000« und einem blinkenden Cursor. Geben Sie nun ein (die Adressen erscheinen automatisch):

```
4000 LDY #00        009 CPY #12
4002 LDA 400E,Y     400B BCC 4002
4005 JSR FFD2       400D BRK
4008 INY
```

Die folgenden Bytes werden wie beschrieben mit einem Punkt eingegeben. Sie werden nicht disassembliert.

```
400E .0D     4017 .54
400F .0D     4018 .20
4010 .53     4019 .53
4011 .4D     401A .55
4012 .4F     401B .50
4013 .4E     401C .45
4014 .20     401D .52
4015 .49     401E .0D
4016 .53     401F .0D
```

Drücken Sie anschließend »F«. Ihr Programm wird noch mal aufgelistet. Starten Sie es nun mit »G 4000«. Es erscheint ein Text auf dem Bildschirm – lassen Sie sich überraschen.

Disassemblieren

Syntax: D [ANFADR,ENDADR]

disassembliert den Bereich von ANFADR bis ENDADR, wobei ENDADR nicht eingegeben werden muß. Wird keine Endadresse eingegeben, erscheint zunächst nur eine Zeile:

ADR	HEXBYTES	BEFEHL
4000	A0 00	LDY #00

Mit der SPACE-Taste wird der jeweils nächste Befehl in der gleichen Art und Weise gezeigt. Wünschen Sie eine fortlaufende Ausgabe, drücken Sie »RETURN«. Die Ausgabe wird dann so lange fortgesetzt, bis eine weitere Taste gedrückt wird oder bis ENDADR erreicht ist. Mit »RUN/STOP« springen Sie jederzeit in den Eingabemodus zurück.

Das Komma, das vor der Adresse auf dem Bildschirm erscheint, ist ein »hidden command« (verstecktes Kommando). Es braucht nicht eingegeben zu werden, da es automatisch beim Disassemblieren angezeigt wird. So ermöglicht es ein einfaches Ändern des Programms. Fahren Sie mit dem Cursor auf den zu ändernden Befehl und überschreiben Sie ihn mit dem neuen. Wenn Sie jetzt »RETURN« drücken, erkennt SMON das Komma als Befehl und führt ihn im Speicher aus. Achten Sie aber darauf, daß der neue Befehl die gleiche Länge (in Byte) hat und füllen Sie gegebenenfalls mit »NOPs« auf. Zur Kontrolle können Sie den geänderten Bereich noch einmal disassemblieren.

Lassen Sie als Beispiel einmal das Programm (siehe Befehl »A«) ab 4000 disassemblieren (»D 4000 4011«). Ändern Sie nun den ersten Befehl auf LDY #01. Die Änderung zeigt sich daran, daß die HEX-Bytes automatisch den neuen Wert annehmen. Starten Sie nun das Programm nochmals mit »G 4000«. Jetzt erscheint der Text mit nur einer Zeile Abstand auf dem Bildschirm.

Starten eines Maschinenprogramms (Go)

Syntax: G [ADRESSE]

startet ein Maschinenprogramm, das bei ADRESSE beginnt. Das Programm muß mit einem BRK-Befehl abgeschlossen werden, damit ein Rücksprung in SMON erfolgen kann. Wird nach »G« keine Adresse eingegeben, benutzt SMON die, die mit dem letzten BRK erreicht worden ist und bei der Register-Ausgabe als PC auftaucht. Mit dem »R«-Befehl (siehe unten) werden die Register vorher auf gewünschte Werte gesetzt.

Memory-Dump

Syntax: M [ANFADR ENDADR]

gibt die HEX-Werte des Speichers sowie die zugehörigen ASCII-Zeichen aus. Auch hier kann auf die Eingabe einer Endadresse verzichtet werden. Die Steuerung der Ausgabe entspricht der beim Disassemblieren.

Beispiel:

M 4000 gibt die Inhalte der Speicherstellen $4000 bis $4007 aus. Weiter geht es wie beim Disassemblieren mit SPACE oder RETURN. Die Bytes können ebenfalls durch

Überschreiben geändert werden, allerdings nicht die ASCII-Zeichen. Verantwortlich dafür ist der Doppelpunkt, der am Anfang jeder Zeile ausgegeben wird, ein weiterer »hidden command«. Wenn Ihre Änderung nicht durchgeführt werden kann, weil Sie zum Beispiel versuchen, ins ROM zu schreiben, wird ein »?« als Fehlermeldung ausgegeben.

Registeranzeige

Syntax: R

zeigt den gegenwärtigen Stand der wichtigsten 6510-Register an: Programmzähler (PC), Status-Register (SR), Akkumulator (AC), X-Register (XR), Y-Register (YR), Stackpointer (SP). Außerdem werden die einzelnen Flags des Status-Registers mit 1 für »gesetzt« und 0 für »nicht gesetzt« angezeigt. Durch Überschreiben werden die Inhalte auf einen gewünschten Wert gesetzt. Die Flags können allerdings nicht einzeln verändert werden, sondern nur durch Überschreiben des Wertes von SR.

Exit

Syntax: X

springt ins Basic zurück. Alle Basic-Pointer bleiben erhalten. Sie können also zum Beispiel direkt im Programm fortfahren, wenn Sie zwischendurch mit SMON einige Speicherstellen kontrolliert haben.

Probieren Sie alle bisher beschriebenen Befehle in Ruhe aus und machen Sie sich mit SMON vertraut. Arbeiten Sie auch parallel den Kurs über Assemblerprogrammierung in dieser Ausgabe durch. Alle Beispiele dort sind auf SMON abgestimmt.

I/O-Set

Syntax: I∅ 1

legt die Device-Nummer für LOAD und SAVE auf 1 (Kassette). Jedes Laden und Abspeichern erfolgt jetzt auf das angegebene Gerät. Die voreingestellte Device-Nummer ist 8 (für die Floppy also: I0 8). Wenn Sie nur mit der Floppy arbeiten, brauchen Sie diesen Befehl also nicht.

LOAD

Syntax: L"name"

lädt ein Programm vom angegebenen Gerät (wie oben beschrieben) an die Originaladresse in den Speicher. Die Basic-Zeiger bleiben bei diesem Ladevorgang unbeeinflußt, das heißt, sie werden nicht verändert.

Beispiel: Unser Monitor soll an seiner Originaladresse ($C000) im Speicher stehen. Also brauchen Sie ihn nur mit »L"SMON"« zu laden, damit er dort erscheint. Wenn Sie einmal ein Programm an eine andere als die Originaladresse laden wollen, dann bietet Ihnen SMON dazu folgende Möglichkeit: »L"name" ADRESSE« lädt ein Programm an die angegebene Adresse. Nehmen Sie doch bitte noch einmal unser letztes Test-Programm und geben es mit dem Assembler ab Adresse $4000 ein. Speichern Sie es mit »S"SUPERTEST" 4000 4023« ab und laden es dann

1. an die Originaladresse (L"SUPERTEST") und
2. an eine andere Adresse (mit L"SUPERTEST"5000 zum Beispiel nach $5000).

Schauen Sie sich danach mit dem Disassembler-Befehl beide Routinen einmal an. Sie werden feststellen, daß beide Programme zwar bis auf die BRANCH-Befehle gleich aussehen, daß das Programm in $5000 aber nicht funktionieren kann, da es eine falsche Adresse verwendet (5002 LDA 400E,Y). Ein anderes Beispiel dazu: Ein Autostart-Programm beginnt bei $0120, läßt sich aber in diesem Bereich nicht untersuchen, da dort der Prozessor-STACK (im Bereich von $0100 bis $01FF) liegt, der vom Prozessor selbständig verändert wird. Wenn Sie nun L"name" 4120 eingeben, befindet sich das Programm anschließend bei $4120 (nicht an der Originaladresse $0120) und Sie können es ohne Einschränkungen – von den falschen Absolut-Adressen abgesehen – disassemblieren.

SAVE

Syntax: S"name", ANFADR ENDADR

speichert ein Programm von ANFADR bis ENDADR-1 unter »name« auf die Floppy ab, da diese – wie wir ja inzwischen wissen – das voreingestellte Gerät ist. Wenn Sie auf Kassette abspeichern wollen, setzen Sie vorher mit »I0 1« die Device-Nummer auf 1.

Beispiel: S"SUPERTEST"4000 4020 speichert das Programm mit dem Namen »SUPERTEST« (es steht im Speicher von $4000 bis $401F) auf Diskette ab. Bitte beachten Sie auch bei diesem Befehl, daß die Endadresse auf das nächste Byte hinter dem Programm gesetzt wird.

Printer-Set

Syntax: P∅ 2

setzt die Primäradresse für den Drucker auf 2. Voreingestellt ist hier die 4 als Gerätenummer (zum Beispiel für Commodore-Drucker). Vielleicht haben Sie es ja schon bemerkt: Bei allen Ausgabe-Befehlen (wie D, M etc.) können Sie auch den Drucker ansprechen, wenn Sie das Kommando geshiftet eingeben. Die Ausgabe erfolgt dann gleichzeitig auf Bildschirm und Drucker. (Beachten Sie bitte die Änderung für die Druckerausgabe am Schluß des Artikels.) Die folgende Befehlsgruppe enthält Befehle zur Zahlen-



umrechnung. Sie wissen ja: Der Mensch mit seinen zehn Fingern neigt eher zur dezimalen Rechenweise, aber der Computer bevorzugt das Binärsystem, weil er nur zwei Finger hat (siehe Netzstecker). Ein Kompromiß ist das Hexade-

Ein bißchen Rechnerei

zimalsystem, denn das versteht keiner von beiden. Um Verständnisschwierigkeiten mit Ihrem Liebling aus dem Weg zu gehen, haben Sie aber SMON.

Umrechnung Dez→Hex

Syntax: # (Dezimalzahl)

rechnet die Dezimalzahl in die entsprechende Hexadezimalzahl um. Hierbei können Sie die Eingabe in beliebiger Weise vornehmen, da SMON Zahlen bis 65535 umrechnet. Beispiel: #12, #144, #3456, #65533 und so weiter.

Umrechnung Hex→Dez

Syntax: $ (Hexadezimalzahl)

rechnet die Hexadezimalzahl in die entsprechende Dezimalzahl um. Die Eingabe muß hierbei zweistellig beziehungsweise vierstellig erfolgen. Ist diese Zahl kleiner als $100 (=255), wird zusätzlich auch der Binärwert ausgegeben.

Beispiel: $12, $0012, $0D, $FFD2 etc. In den ersten drei Beispielen erfolgt die Anzeige auch in binärer Form.

Umrechnung Binär→Hex,Dez

Syntax: % (Binärzahl (achtstellig))

rechnet die Binärzahl in die entsprechenden Hexa- und Dezimalzahlen um. Bei diesem Befehl müssen Sie genau acht Binärzahlen eingeben. Falls Sie einmal versehentlich mehr eingeben sollten, werden nur die ersten acht zur Umrechnung herangezogen. Beispiel: %00011111, %10101011

Add-Sub

Syntax: ? 2340+156D

berechnet die Summe der beiden vier (!)-stelligen Hex-Zahlen. Neben der Addition ist auch Subtraktion möglich.

Programme auf dem Rangierbahnhof

Occupy (Besetzen)

Syntax: O (ANFADR ENDADR HEX-Wert)

belegt den angegebenen Bereich mit dem vorgegebenen HEX-Wert. Beispiel: O 5000 8000 00 füllt den Bereich von $5000 bis $7FFF mit Nullen.

Man kann mit »OCCUPY« aber nicht nur Speicherbereiche löschen, sondern auch mit beliebigen Werten belegen. Häufig hat man das Problem, festzustellen, welcher Speicherplatz von einem Programm wirklich benutzt wird. Wir füllen den in Frage kommenden Bereich dann zuerst zum Beispiel mit »AA« und laden dann unser Programm. Probieren Sie bitte das folgende Beispiel: Füllen Sie den Speicherbereich von $3000 bis $6000 mit $AA und laden Sie dann unser SUPERTEST-Programm. Beim Disassemblieren können Sie erkennen, daß unser kleines Programm exakt zwischen vielen »AA« eingebettet ist.

Write

Syntax: W (ANFADRalt ENDADRalt ANFADRneu)

verschiebt den Speicherbereich von ANFADRalt bis ENDADRalt nach ANFADRneu ohne Umrechnung der Adressen! Unser kleines Testprogramm möge noch einmal als Beispiel dienen:

W 4000 4020 6000 verschiebt das oben angesprochene Programm von $4000 nach $6000.

Hierbei werden weder die absoluten Adressen umgerechnet noch die Tabellen geändert. Letzteres ist sicherlich erwünscht, aber denken Sie daran, daß das verschobene Programm nun nicht mehr lauffähig ist, da die absoluten Adressen nicht mehr stimmen (zum Beispiel bei dem Befehl LDA 400E,Y). Falls Sie jetzt »G6000« eingeben, um das Programm zu starten, werden Sie sich sicherlich wundern, daß es dennoch läuft. Doch löschen Sie einmal das Programm in $4000 (mit »O4000 4100 AA«) und starten das Programm in $6000 noch einmal! Seltsam, nicht? Abhilfe schafft der nächste Befehl.

Variation

Syntax: V (ANFADRalt ENDADRalt ANFADRneu ANFADR ENDADR)

rechnet alle absoluten Adressen im Bereich von ANFADR bis ENDADR, die sich auf ANFADRalt bis ENDADRalt beziehen, auf ANFADRneu um. Kompliziert? Nicht, wenn Sie sich klarmachen, daß die ersten drei Adressen exakt den Eingaben beim »W«-Befehl entsprechen. Neu hinzu kommen nur die beiden Adressen für den Bereich, in dem die Änderung tatsächlich erfolgt.

Um unser mit »W« schon verschobenes Programm auch wieder lauffähig zu machen, geben Sie folgendes ein: V 4000 4020 6000 6000 600E. Damit werden alle Absolutadressen, die im Bereich von $6000 bis $600E – dahinter steht die Tabelle – liegen und sich bisher auf $4000 bis $4020 bezogen haben, auf den neuen Bereich umgerechnet. Probieren geht wie immer über kapieren.

Eine Zusammenfassung dieser beiden Befehle ermöglicht:

Convertieren

(Verschieben eines Programmes mit Adreßumrechnung.)

Syntax: C (ANFADRalt ENDADRalt ANFADRneu ANFADRges ENDADRges)

verschiebt das Programm von ANFADRalt bis ENDADRalt zur ANFADRneu, und zwar mit Umrechnung der Adressen zwischen ANFADRges und ENDADRges

An unserem kleinen Testprogramm läßt sich wieder einmal demonstrieren, wie der Befehl eingesetzt wird. Laden Sie es also mit »L"SUPERTEST"« und schauen es mit »D 4000« an. Jetzt wollen wir an der Adresse $4008 einen 3-Byte-Befehl einfügen: C 4008 4020 400B 4000 4011 verschiebt das Programm von $4008 bis $4020 zur neuen Anfangsadresse $400B. Dabei werden im Bereich von $4000 bis $4011 (neue Endadresse des »aktiven« Programmes!) die Sprungadressen umgerechnet. Nun können Sie ab Adresse $4008 einen 3-Byte-Befehl einfügen, zum Beispiel STY 0286. Dazu geben Sie bitte ein:

```
A 4008
4008 STY 0286
F
```

Überzeugen Sie sich davon, daß SMON die Befehle korrekt umgerechnet hat, indem Sie unser Beispiel disassemblieren (D 4000) und anschließend mit G 4000 starten. Besitzer eines Farbmonitors werden in helle Begeisterung ausbrechen. Vorsicht ist geboten, wenn Tabellen oder Text vorhanden sind. SMON wird versuchen, diese als Befehle zu disassemblieren und gegebenenfalls umzurechnen. Dabei können unvorhersehbare Verfälschungen auftreten. Aus diesem Grunde ist im Beispiel die Endadresse des zu ändernden Bereiches auf $4011 und nicht etwa auf $4023

BASIC-DATA

gelegt worden. Wenn Sie größere Programme zu verschieben haben, sollten Sie die Kommandos W und V anwenden beziehungsweise einen Assembler einsetzen (zum Beispiel Hypra-Ass), der es Ihnen gestattet, beliebige Einfügungen, Verschiebungen und sonstige Änderungen vorzunehmen. Das C-Kommando eignet sich in erster Linie für kleinere Änderungen innerhalb eines Programms.

B (Anfadr Endadr)
wandelt das Maschinenprogramm von ANFADR bis ENDADR-1 in Basic-DATA-Zeilen um.
B 4000 4020

Unser Testprogramm wird in DATA-Werte umgerechnet und dann mit Zeilennummer 32000 beginnend im Basic-Speicher abgelegt. Ein im Speicher befindliches Basic-Programm (zum Beispiel ein Basic-Lader) mit kleineren Zeilennummern kann dann diese DATA-Zeilen benutzen.

Wenn Sie das Testprogramm wie oben beschrieben umgewandelt haben, überzeugen Sie sich mit »LIST« von der Ausführung. Dann können Sie folgendes eingeben:
10 FOR I=16384 TO 16415 : READ D :POKE I,D : NEXT

In Verbindung mit den oben erzeugten DATA-Zeilen (und RUN!) hätten Sie wieder das ursprüngliche Maschinenprogramm im Speicher. Falls Sie dieses Beispiel durchführen wollen, denken Sie bitte daran, daß Sie nach Erstellung der DATAs das Originalprogramm zum Beispiel mit OCCUPY (O 4000 4020 AA) überschreiben, damit Sie die richtige Ausführung überprüfen können.

Der BRK-Befehl am Ende des Testprogramms bewirkt einen Sprung zum SMON zurück. Wollen Sie ein Maschinenprogramm von Basic aus starten und auch wieder dorthin zurückgelangen, muß der letzte Befehl ein RTS sein. Probieren Sie es aus, indem Sie das Basic-Programm um 20 SYS 16384 erweitern.

KONTROLLE

K (Anfadr Endadr)
listet die ASCII-Zeichen im gewünschten Bereich. Es werden jeweils 32 Zeichen pro Zeile ausgegeben, so daß man sich einen schnellen Überblick über Texte oder Tabellen verschaffen kann.

Beispiel: K 4000 listet die ersten 32 Zeichen unseres Programms. Die weitere Ausgabe ist genau wie beim Disassemblieren durch Druck auf SPACE oder RETURN möglich. Auch hier können Sie wie bei den anderen Bildschirm-Ausgabebefehlen Änderungen durch einfaches Überschreiben vornehmen (natürlich nicht im ROM und nur mit ASCII-Zeichen!).

Als Beispiel wollen wir einmal im Basic »herumpfuschen«. Das geht natürlich nicht so ohne weiteres, weil das Basic im ROM steht und damit nicht verändert werden kann. Tippen Sie bitte folgendes ein:
W A000 C000 A000

Auf den ersten Blick eine unsinnige Anweisung; der Speicher soll von A000 bis C000 nach A000 verschoben werden. Dieser Befehl entspricht exakt der Basic-Schleife
FOR I = 40960 TO 49152 : POKE I, PEEK (I) : NEXT

Nun ist es aber so, daß beim PEEK das ROM gelesen, beim POKE aber ins darunterliegende RAM geschrieben wird. Wir erreichen also, daß das Basic ins RAM kopiert wird. Jetzt müssen wir dafür sorgen, daß das Betriebssystem sein Basic aus dem RAM und nicht aus dem ROM holt. Zuständig dafür ist die Speicherstelle 0001. Geben Sie bitte »M 0001« ein und überschreiben Sie die »37« mit »36«.

Es passiert gar nichts. Jetzt tritt unser K-Kommando in Aktion. Geben Sie ein: K A100 A360

Was Sie sehen, sind die Basic-Befehlswörter und -Meldungen. Schalten Sie mit SHIFT/CBM auf Kleinschrift, dann erkennen Sie, daß der jeweils letzte Buchstabe eines Befehlswortes groß geschrieben ist (Endekennung). Jetzt ändern Sie durch Überschreiben das »LIST« (A100) in »LUST« und »ERROR« (A360) in »FAELER«. (Bei »FAELER« müssen Sie ein Zeichen vor »ERROR« beginnen, sonst paßt es nicht.)

Verlassen Sie jetzt SMON mit »X« und geben Sie danach ein:
POKE 1,54

SMON schaltet nämlich beim »X«-Befehl immer auf das Basic-ROM zurück, daher müssen wir wieder auf unser geändertes Basic umschalten. Schreiben Sie nun einen Basic-Dreizeiler und versuchen Sie, diesen zu LISTen. Ergebnis? Versuchen Sie es jetzt einmal mit »LUST«. Ihrer weiteren Phantasie sind keine Grenzen mehr gesetzt...

FIND

Wie oben angesprochen stellt SMON eine Reihe verschiedener Suchroutinen zur Verfügung, die im folgenden an vielen Beispielen beschrieben werden. Alle diese Befehle bestehen aus zwei Zeichen und beginnen mit dem Buchstaben »F«.

F (HEX-WERT(e), Anfadr Endadr)
sucht nach einzelnen HEX-Werten innerhalb eines bestimmten Bereichs. Das zweite Zeichen (hinter F) ist hier ein Leerzeichen und darf nicht weggelassen werden! Die Bereichsangabe kann wie bei allen folgenden Befehlen entfallen, dann wird der gesamte Speicher durchsucht.

Beispiel: Wir suchen alle Befehle LDY #01, also die Werte A0 01 im Bereich von $2000 bis $6000.

F A0 01, 2000 6000 (die Leerzeichen zwischen den Hex-Bytes dürfen nicht weggelassen werden!). Es erscheinen alle Speicherstellen, die die gesuchten Bytes enthalten, also zum Beispiel 4000.

FA (Adresse, Anfadr Endadr)
sucht alle Befehle, die eine bestimmte Adresse als Operanden haben (absolut). Die Adresse braucht nicht vollständig angegeben zu werden, es kann das Jokerzeichen »*« benutzt werden.

1. Beispiel: Wir suchen alle JSR FFD2-Befehle im Bereich $2000 bis $6000.
FAFFD2,2000 6000

Es erscheinen alle Befehle disassembliert, die FFD2 im Operanden enthalten (also auch LDA FFD2 oder STA FFD2,Y...).

2. Beispiel: Wir suchen alle Befehle, die auf den I/O-Bereich ($D000 bis $DFFF) zugreifen.
FAD***,2000 6000

Der Joker kann aber auch zum Beispiel zur Suche im Bereich $D000 bis $D0FF dienen: FAD0**,2000 6000

FR (Adresse, Anfadr Endadr)
sucht nach relativen Sprungzielen. Anders als bei absoluten Sprüngen (JMP, JSR) benutzen die Branch-Befehle eine relative Adressierung, also zum Beispiel »Verzweige 10 vor« oder »37 zurück«. Solche Sprünge lassen sich mit dem FA-Kommando nicht finden. Hier wird »FR« eingesetzt.

Beispiel: Gesucht werden alle Branch-Befehle, die die Adresse $4002 anspringen.
FR4002,2000 6000

Natürlich können solche Befehle nur höchstens 128 Byte vom Sprungziel entfernt sein. Die Bereichsangabe ist hier also viel zu groß gewählt (SMON stört dies allerdings nicht). Der Einsatz des Jokers ist hier ebenfalls wie oben beschrieben möglich.

FT (Anfadr Endadr)
sucht Tabellen im angegebenen Bereich. SMON behandelt dabei alles, was sich nicht disassemblieren läßt, als Tabelle.
Beispiel: Wir suchen Tabellen oder Text im Bereich $2000 bis $6000.
FT 2000 6000

FZ (Adr, Anfadr Endadr)
sucht alle Befehle, die Zeropage-Adressen haben.

1. Beispiel: FZC5,2000 6000 findet alle Befehle, die C5 adressieren, also zum Beispiel BIT $C5, LDA (C5), Y etc.

2. Beispiel: FZF*,2000 6000 findet alle Befehle, die den Bereich zwischen $F0 und $FF adressieren.

3. Beispiel: FZ**,2000 6000 findet sämtliche Befehle mit Zeropage-Adressierung.

FI (Operand, Anfadr Endadr)
sucht alle Befehle mit unmittelbarer Adressierung (immediate).

Beispiel: Gesucht werden Befehle, die zum Beispiel das Y-Register mit 01 laden. FI01,2000 6000 findet LDY #01 in Adresse $4000.

Sie sehen, SMON bietet eine Fülle von verschiedensten FIND-Routinen, mit denen alles gesucht und auch gefunden (!) werden kann.

= Adr1 Adr2
Vergleichen von Speicherstellen
= 4000 6000
vergleicht den Speicherinhalt ab $4000 mit dem ab $6000. Das erste nicht übereinstimmende Byte wird angezeigt und der Vergleich wird abgebrochen.

Wenn Sie ein Maschinenprogramm geschrieben und überarbeitet haben und Sie wissen nicht mehr, worin eigentlich der Unterschied zwischen der 76. und der 77. Version besteht, gehen Sie so vor: Laden Sie zuerst Version 76 und verschieben Sie diese mit dem »W«-Befehl in einen freien Speicherbereich. Laden Sie dann Version 77 und führen Sie den »=«-Befehl durch. Sofort finden Sie den Unterschied und können mit der Arbeit an Version 78 beginnen ...

Trace: Schritt für Schritt

Wir wollen uns bei der Beschreibung der Trace-Befehle auf Anwendungsbeispiele konzentrieren. Zum Aufbau der Routine sei nur so viel gesagt: Gesteuert wird sie mit Hilfe des Prozessor-Interrupts, weil nur damit ein Eingriff ins laufende Maschinenprogramm möglich ist. Während des Trace-Ablaufs wird deswegen der Bildschirm kurzfristig aus- und eingeschaltet, weil alle anderen Interruptanforderungen, wie zum Beispiel durch den Video-Chip, verhindert werden müssen. Da die Befehle eines Programms nicht nur angezeigt, sondern auch wirklich ausgeführt werden, ist der »SEI«-Befehl mit großer Vorsicht zu verwenden. Doch dazu später mehr. Wir wollen ein neues, besser geeignetes Beispiel verwenden als bisher. Tippen Sie also das folgende Miniprogramm mit dem Assembler ein (A 4000):

```
4000   LDA   #30    lade den Akku mit (ASCII-) 0
4002   JSR   FFD2   gib Akku auf dem Bildschirm aus
4005   CLC
4006   ADC   #01    erhöhe Akku um 1
4008   CMP   #39    vergleiche Akku mit (ASCII-) 9
400A   BCC   4002   springe, wenn Akku kleiner, zurück
400C   BRK          springe in SMON zurück
```

Starten Sie das Programm mit »G 4000«. Es muß die Zahlen von 0 bis 8 auf den Bildschirm schreiben.

Trace-Stop
TS (Startadresse Stoppadresse)
Starten Sie nun unser Programm mit TS 4000 4009. Die ersten Befehle werden ausgeführt (die Null ausgegeben, der Akku erhöht etc.), dann stoppt das Programm bei Adresse $4009 und springt in die Registeranzeige.

Genau genommen ist »TS« gar kein Trace-Befehl, das Programm läuft nämlich bis zur gewählten Stoppadresse in Echtzeit durch. Dort angekommen, können Sie die Register prüfen und gegebenenfalls durch Überschreiben ändern. Mit »G«, »TW« oder »TB« (wird später erklärt) ohne weitere Adresseneingaben können Sie dann im Programmlauf fortfahren. SMON merkt sich nämlich, wo er stehengeblieben ist und arbeitet ab dieser Adresse weiter, wenn Sie nicht eine neue angeben.

Sinnvoll ist dieser Befehl immer dann, wenn in einem längeren Programm nur bestimmte Teile »getraced« werden sollen, der Anfang aber durchlaufen werden muß, um Variable zu setzen oder Benutzereingaben zu erfragen. Auch wenn man nicht ganz sicher ist, ob eine bestimmte Passage überhaupt jemals durchlaufen wird, kann man das mit »TS« überprüfen.

Zwei Einschränkungen gibt es allerdings wegen der Arbeitsweise dieses Befehls: SMON setzt im Programm an die Stoppadresse einen BRK-Befehl und merkt sich, welcher Befehl dort stand, um ihn wieder zurückzuschreiben. Deshalb funktioniert »TS« nur im RAM, nicht aber zum Beispiel im Basic oder im Betriebssystem. Auch darf die Speicherstelle, in der sich SMON den ausgetauschten Befehl merkt ($02BC) vom Programm nicht verändert werden, sonst ist eine korrekte Reparatur nicht mehr möglich.

Der wohl am häufigsten und vielseitigsten eingesetzte Trace-Befehl ist sicherlich »TW«.

Trace Walk
TW (Startadresse)
Starten Sie unser Beispiel jetzt mit TW 4000

Der erste Befehl (LDA #30 in Adresse $4000) wird ausgeführt, SMON stoppt und zeigt dann die Inhalte aller Register in der gleichen Reihenfolge wie beim »R«-Kommando sowie den nächsten Befehl an. Im Akku steht jetzt 30, der Programmzähler zeigt auf $4002. Jetzt drücken Sie eine Taste. Der nächste Befehl (JSR FFD2) wird ausgeführt, der Programmzähler zeigt auf $FFD2. Achten Sie auf den Stackpointer: Sein Inhalt hat sich um 2 vermindert, weil der Prozessor auf dem Stack die Adresse abgelegt hat, an die er nach Beendigung der Subroutine zurückspringen soll. Der nächste angezeigte Befehl ist ein indirekter Sprung über $0326. Mit dem nächsten Tastendruck wird er durchgeführt.

Und so geht es munter weiter. Verzweifeln Sie nicht, wenn Sie auch nach den nächsten zehn Tastendrücken immer noch irgendwo im Betriebssystem »herumtracen« und von unserem Beispielprogramm weit und breit nichts mehr zu sehen ist. Ausnahmsweise ist unser Liebling einmal nicht im »Land der Träume« verschwunden, sondern tut, was er soll: Er arbeitet brav einen Befehl nach dem anderen ab, der zur Routine $FFD2 gehört, und das ist reichlich viel. Also bewegen Sie Ihre Finger, Sie haben's ja nicht anders gewollt. Irgendwann einmal, nach mehreren hundert gedrückten Tasten, befinden Sie sich plötzlich wieder in der Registeranzeige von SMON. Das Programm ist beendet. Nun werden Sie enttäuscht fragen, was man wohl mit einem Trace-Modus anfangen soll, der schon bei kleinsten Beispielprogrammen ein völlig undurchschaubares Chaos erzeugt? Nur Geduld, die Rettung naht in Gestalt der Taste <J>.

Falls Ihre Hand noch nicht in Gips liegt, starten Sie das Ganze noch mal von vorn mit »TW 4000«. Diesmal drücken Sie aber jedesmal, wenn als nächster Befehl »JSR FFD2« angezeigt wird, auf <J>. Der Effekt ist, daß die gesamte Subroutine auf einen Schlag abgearbeitet wird und Sie sofort wieder auf dem nächsten Befehl unseres Beispiels landen. Daß wir nicht gemogelt und die Befehle von »JSR FFD2« einfach unterschlagen haben, sehen Sie daran, daß der Akku tatsächlich auf dem Bildschirm ausgegeben worden ist (rechts neben FFD2). Jetzt können Sie unser Beispiel in aller Ruhe bis zum Ende durchgehen und verfolgen, wie der Akku erhöht wird, wie der Vergleich das Status-

register beeinflußt und wie entsprechend der Rücksprung in die Schleife erfolgt.

Sie dürfen die <J>-Taste auch dann benutzen, wenn Sie schon mitten in der Subroutine sind. Aber hierbei ist äußerste Vorsicht geboten: Die Rücksprungadresse muß unbedingt oben auf dem Stack liegen, wenn Sie »J« drücken. Hat nämlich der Prozessor Werte auf dem Stack abgelegt (mit PHA oder PHP), dann erfolgt der Sprung irgendwo hin, nur nicht zurück ins Programm. Achten Sie deshalb genau auf die Anzeige des Stackpointers. Wenn dessen Wert genau so groß ist wie bei Beginn der Subroutine, kann nichts passieren. Sonst hilft nur noch der Reset-Taster, den Sie ja inzwischen hoffentlich eingebaut haben, oder eine ruhige Hand, die die Büroklammer an Pin 1 und 3 des User-Ports hält (Kostenpunkt der Reparatur bei Abrutschen liegt bei zirka 100 Mark ...).

»TW« bricht automatisch mit der Registeranzeige ab, wenn im Programm ein »BRK«-Befehl auftaucht. Wenn Ihnen das zu lange dauert oder Sie zwischendurch ein Register ändern möchten, können Sie den Trace-Modus jederzeit mit der Stopp-Taste verlassen. Anschließend können Sie wie bei »TS« beschrieben fortfahren.

Im Gegensatz zu »TS« können Sie mit »TW« auch im ROM herumstöbern; Sie haben es ja bei der Subroutine $FFD2 bereits getan. Einzige Einschränkung beim »TW«-Befehl: Ihr Programm darf keinen »SEI« enthalten, da dieser den Interrupt und damit auch den Trace-Modus lahmlegt. Verlassen Sie in diesem Falle »TW« mit STOP und starten erneut hinter dem »SEI«-Befehl. Allerdings müssen Sie in Kauf nehmen, daß das Programm normalerweise nicht mehr korrekt arbeitet.

Das nächste Programm soll als weiteres Beispiel für den TW-Modus dienen. Geben Sie es folgendermaßen ein:

```
5000  LDA  #00   lädt den Akku mit »0«
5002  TAX        überträgt den Akku ins X-Register
5003  .0C        ein mysteriöses Byte
5004  LDA  #04   lädt den Akku mit »4«
5006  TAY        überträgt den Akku ins Y-Register
5007  BRK        springt in SMON
```

Wenn wir dieses kleine Programm abarbeiten, müßte das X-Register auf »0« stehen, während Akku und Y-Register mit »4« geladen sind. Starten wir also das Programm mit »G 5000« und schauen uns die Register an.

Seltsamerweise enthalten alle Register eine »0«. Vorsichtig, wie wir sind, überschreiben wir die drei Register mit »FF«, um die Veränderung deutlich kontrollieren zu können.

Dann starten wir mit »G 5000« ein zweites Mal. Gegen alle Gesetze der Vernunft erscheint wieder das »falsche« Ergebnis – alle drei Register sind »0«. Hier soll uns jetzt der TW-Modus weiterhelfen, indem er uns zeigt, was in Wirklichkeit passiert.

Geben wir »TW 5000« ein. Der erste Befehl (LDA #00) ist durchgeführt, im Akku erscheint die Null. Jetzt steht der Programmzähler auf dem folgenden Befehl »5002 TAX«. Nach Drücken einer Taste wird dieser Befehl ausgeführt und es erscheint die Null im X-Register. Beim folgenden Befehl müssen wir feststellen, daß der Disassembler nicht in der Lage ist, ihn zu interpretieren – er gibt drei Sternchen aus. Hierbei handelt es sich um unser Byte »0C«.

Wieder ein Tastendruck; und dann erkennen wir, daß etwas Merkwürdiges passiert ist. Der Prozessor hat augenscheinlich den nächsten Befehl (LDA #04) übersprungen und steht schon auf dem folgenden »TAY«. So also wird unser Programm abgearbeitet. Damit ist auch das »falsche« Ergebnis erklärt. Bleibt nur noch die Frage nach dem Grund für dieses seltsame Verhalten. Und der ist sicherlich in dem mysteriösen Byte »OC« zu suchen. Hierbei handelt es sich um einen der »inoffiziellen« Opcodes, die aufgrund der Prozessorarchitektur vorhanden sind und in manchen Programmen ihr Unwesen treiben – wie wir zu unserem Leidwesen erfahren mußten. Das Byte »OC« wirkt wie ein »NOP«, der eine Länge von 3 Byte hat. Deshalb wird der folgende 2-Byte-Befehl (LDA #04) verschluckt.

Es gibt noch einiges zu entdecken am 6502 und 6510 – TW macht's möglich.

Häufig ist es nicht sinnvoll, ein Programm von Anfang an im TW-Modus laufenzulassen. Zum anderen sind gerade Schleifen, die per Hand mit »TW« durchlaufen werden müssen, eine ermüdende Angelegenheit. Hier bietet SMON neben dem bereits beschriebenen »TS« eine weitere Trace-Möglichkeit an:

Trace Break

TB (Adresse Anzahl der Durchläufe)

Trace Quick

TQ (Adresse)

Geben Sie als Beispiel folgendes Programm ein:

```
6000  LDY  #00     Y als Zähler auf »0«
6002  LDA  600E,Y  Werte von $600E ff. sollen geladen
                   werden
6005  JSR  FFD2    Ausgabe der Zeichen auf dem Bild-
                   schirm
6008  INY          der Zähler wird erhöht
6009  CPY  #0E     Zähler schon »14«?
600B  BCC  6000    wenn nein, dann nächsten Wert holen
601D  BRK
```

Bei $600E soll nun ein Text stehen, den das Programm ausgibt. Die einfachste Art, mit SMON Texte in den Speicher zu schreiben, besteht im »K«-Befehl. Geben Sie

K 600E

ein (danach natürlich Return) und drücken Sie die STOP-Taste. Fahren Sie mit dem Cursor an das erste ausgegebene Zeichen (vermutlich ein Punkt) und schreiben Sie – ohne Anführungszeichen:

»FEHLER BEHOBEN«

Drücken Sie dann Return, um die Zeile an den Rechner zu übergeben. Wenn Sie das Programm starten, werden Sie wieder einmal Gelegenheit haben, sich in Ruhe etwas zu trinken zu holen (Prost!), denn das Programm enthält einen dummen Fehler und beschäftigt den Computer für eine lange, lange Zeit. Genauer gesagt, bis Sie ihn mit Reset (zum Beispiel durch RUN/STOP-RESTORE) erlösen.

Nun soll SMON helfen, diesen Fehler zu lokalisieren. Setzen Sie zuerst einmal einen Breakpoint bei $6002 und begrenzen die Durchläufe auf die maximale Anzahl:

TB 6002 0E

und starten mit

TQ 6000

den Quicktrace bei $6000. Das Programm läuft so lange, bis zum 14. Mal die Adresse $6002 erreicht wird und springt dann in den TW-Modus. Wenn Sie sich jetzt die Registerinhalte genau anschauen, müßte Ihnen der Fehler geradezu ins Auge springen. Wie groß sollte denn das Y-Register sein? Welchen Wert sollte der Akku haben? NA?!

Wenn Sie Programme mit SMON untersuchen oder verändern wollen, müssen Sie noch wissen, welche Speicherstellen SMON verwendet. Es soll ja Monitorprogramme geben, die die Basic-Zeiger als Arbeitsspeicher benutzen, so daß ein Basic-Programm nach dem Rücksprung aus dem Monitor gelöscht ist. SMON tut so etwas nicht. Aber natürlich braucht er auch Speicherstellen, um sich Werte mer-

Das »Gedächtnis« von SMON

ken zu können. Damit Sie Konflikten von Anfang an aus dem Wege gehen können, sind die wichtigsten hier dargestellt.

In der Zeropage belegt SMON den Bereich von $00A4 bis $00B6. Dort stehen Systemvariable für die Kassettensicherung und die RS232-Schnittstelle. Diese werden nur während des Betriebs der Kassette oder von RS232 gebraucht, sind ansonsten aber frei. Außerdem werden die Speicherstellen $00FB bis $00FF benutzt, die sowieso zur freien Verfügung des Anwenders vorgesehen sind. Alle anderen Zeiger in der Zeropage, also insbesondere die Speicherverwaltung für Basic bleiben unbeeinflußt.

Als weiteren Arbeitsspeicher benutzt SMON den Bereich von $02A8 bis $02C0. Auch dieser Bereich wird vom Betriebssystem nicht benutzt, so daß keine Konflikte entstehen dürften. Beim Assemblieren wird zusätzlich noch der Kassettenpuffer als Speicher für die Label benötigt. Dieser bleibt ansonsten aber auch unverändert; das ist wichtig, wenn Maschinenroutinen dort abgelegt werden sollen.

Alles in allem ist SMON also recht verträglich.

SMON verschieben? – Mit SMON!

Eine Reihe von Anfragen hat uns erreicht, ob man SMON nicht mit Hilfe des »W«-, »V«- oder »C«-Kommandos verschieben könne. Alle Versuche in dieser Richtung seien fehlgeschlagen. Einige Leser meinten auch, in der V-Routine müsse ein Fehler stecken. Diesmal sind wir jedoch völlig schuldlos; es gibt nämlich einige Befehle in SMON, die keine Sprungadressen sind und sich trotzdem auf den Bereich ($C000-) beziehen, in dem SMON steht.

Dazu gehören in erster Linie die oben erwähnten Einsprungadressen, deren High-Byte natürlich geändert werden muß, wenn SMON in einem anderen Speicherbereich laufen soll. Es gibt aber auch Befehle, die eine Adresse im Programm in einem Vektor ablegen müssen. Disassemblieren Sie einmal den Anfang von SMON mit »D C000 C00B«. Sie erhalten

```
LDA   #14    Low-Byte der BREAK-Routine von SMON
STA   0316   im Break-Vektor speichern
LDA   #C2    High-Byte (!) siehe oben
STA   0317   siehe oben
BRK
```

Damit wird der Break-Vektor des Betriebssystems auf den SMON gesetzt und mit dem anschließenden — und jedem weiteren BRK-Befehl — springt das Programm in SMONs BREAK-Routine. Wenn SMON in einem anderen Bereich als $C000 laufen soll, dann müssen diese Befehle geändert werden.

Heraussuchen kann man sie mit »FIC*,C000 D000«. Sie wissen doch noch, was diese Anweisung bedeutet: Suche mir alle Befehle, die ein Register unmittelbar mit einem Wert laden, der mit $C beginnt. Aber Vorsicht! Nicht alles, was da angezeigt wird, muß auch geändert werden! Um Ihnen weitere Stunden sinnlosen Herumbrütens zu ersparen, wollen wir als Beispiel zeigen, wie man SMON in den Bereich $9000 bis $A000 verlegen kann. Natürlich geht das im Prinzip für jeden anderen Bereich genauso; wir selbst haben insgesamt fünf SMON-Versionen für fünf verschiedene Speicherbereiche, von denen eine immer paßt.

1. Wir verschieben zuerst das ganze Programm ohne Umrechnen in den neuen Bereich:
W C000 CFFA 9000

2. Nun lassen wir alle absoluten (3-Byte-)Befehle umrechnen. Die Tabellen am Anfang von SMON bleiben verschont:
V C000 CFFA 9000 920B 9FD2

3. Als nächstes ändern wir die High-Bytes der Befehlsadresse. Geben Sie
»M 902B 906B«
ein und ändern Sie in jedem zweiten Byte das »C« durch Überschreiben in »9«. Vergessen Sie nicht, am Ende jeder Zeile »RETURN« zu drücken, damit Ihre Änderung auch übernommen wird.

4. Nun sind die Befehle mit Immediate-Adressierung an der Reihe. Sie müssen so geändert werden, daß sie sich auf den neuen Bereich $9... beziehen. Suchen Sie sie mit
FIC*,9000 9FFA
heraus. Sie erhalten

```
9005  LDA  #C2  ändern
9124  CPX  #C0  nicht ändern
9386  LDY  #C0  ändern
9441  CMP  #C0  nicht ändern
987F  LDX  #C3  nicht ändern
988D  LDX  #C1  nicht ändern
9992  LDA  #C1  nicht ändern
9C2C  LDA  #CC  ändern
9C5B  LDA  #C2  ändern
9CF4  LDA  #CC  ändern
9DA1  LDX  #CC  ändern
9E03  LDA  #CC  ändern
9E6C  CMP  #C0  nicht ändern
9F71  LDY  #CF  ändern
```

Sie sehen, es gibt keine Regel, welche Befehle zu ändern sind und welche nicht. Aus diesem Grunde müssen Sie diese Änderungen »von Hand« vornehmen.

5. Die Adressen im Diskmonitor müssen ebenfalls umgestellt werden. Dazu geben Sie bitte ein:
M 9FD8 9FE4
und ändern Sie jedes zweite Byte wie unter Punkt 3 beschrieben.

Vergessen Sie bitte auf keinen Fall, Ihre neue(n) Version(en) unter neuem Namen zu speichern. Sie lassen sich dann mit LOAD "Name",8,1 von Diskette laden und mit dem entsprechenden SYS (zum Beispiel 36864 bei SMON $9000) starten. Denken Sie auch daran, nach dem Laden und vor dem SYS ein NEW einzugeben, sonst beschwert sich der B-Befehl mit einem OUT OF MEMORY ERROR.

Probieren Sie nun alle Befehle durch. Sie müssen genauso arbeiten wie bisher. Vor allem können Sie jetzt auch Programme wie »DOS 5.1« oder »Turbo Tape« untersuchen, die im $C000-Bereich stehen. Achten Sie aber, wenn Sie »SMON $9000« von Basic aus benutzen, darauf, daß das Basic ihn nicht überschreibt. String-Variable werden nämlich von $A000 nach unten hin aufgebaut und bis $9E09 ist nicht viel Platz. Schützen Sie im Zweifelsfalle den Bereich, indem Sie nach dem Laden des SMON $9000 eingeben:

NEW : POKE 56,144 : POKE 55,0

Damit ist SMON vor Überschreiben geschützt. Das ist natürlich bei dem SMON $C000 nicht nötig, weil Basic in diesen Bereich nicht hineinkommt.

Die Befehle des Disk-Monitors

Da das Arbeiten mit dem Disk-Monitor besondere Aufmerksamkeit verlangt (nach Murphys Gesetzen führen Fehleingaben in der Regel zu unlesbaren Disketten), wird er mit einem eigenen Kommando eingeschaltet. Leider waren alle halbwegs sinnvollen Buchstaben (»D« wie Diskette oder »F« wie Floppy) schon vergeben, deshalb haben wir uns für ein schlichtes »Z« wie Zuversicht entschieden.

-Z schaltet den Disk-Monitor ein.

Die Rahmenfarbe ändert sich auf Gelb, der gewohnte ».« am Anfang einer Zeile ändert sich in »*«. Dies alles hat den

Zweck, Ihnen deutlich zu machen, daß es jetzt ernst wird. Intern wird jetzt das Basic abgeschaltet, weil der Disk-Monitor einen 256 Byte großen Puffer benötigt. Dieser liegt von $BF00 bis $C000 im RAM unter dem Basic, weil er dort am wenigsten stören kann.

READ: R (Track Sektor)

Liest einen Block von der Diskette in den Computer. Track und Sektor müssen als Hexzahlen eingegeben werden. Die erste Zeile des Blocks wird ausgegeben. Da wir dazu normale SMON-Routinen verwenden, steht als Speicheradresse $BF00. Das »BF« können Sie vorerst ignorieren. Die weitere Ausgabe des Hexdump erfolgt anders als gewohnt mit der Taste »SHIFT«. ÷STOP° bricht die Ausgabe ab. Sie können die Hex-Bytes überschreiben und damit ändern. Eine dauerhafte Änderung erfolgt aber erst beim Zurückschreiben auf die Diskette (siehe Befehl »W«). Geben Sie nur »R« ohne Track und Sektor ein, wird der logisch (!) nächste Block eingelesen.

MEMORY-DUMP: M

Zeigt den gerade im Puffer befindlichen Block nochmals auf dem Bildschirm an.

Genau wie beim R-Befehl können Sie die Ausgabe mit »SHIFT« und »STOP« steuern und Änderungen vornehmen.

WRITE: W (Track Sektor)

Schreibt einen Block aus dem Puffer auf die Diskette zurück. Ähnlich wie bei »R« kann die Angabe von Track und Sektor entfallen. Es wird dann der Track und Sektor des letzten R-Befehls benutzt. Das ist in fast allen Fällen auch der richtige.

ERROR: @

Liest den Fehlerkanal aus, gibt ihn aber nur aus, wenn wirklich ein Fehler vorhanden war. (»00, OK, 00, 00« wird unterdrückt.)

EXIT: X

Verläßt den Disk-Monitor und springt in den SMON zurück. Dabei wird die Rahmenfarbe auf Blau zurückgeschaltet und es erscheint wieder der ».« am Anfang der Zeile. Das Basic wird wieder eingeschaltet. Wollen Sie nun mit SMON-Kommandos auf den Puffer zugreifen, müssen Sie Basic wieder abschalten ($36 in Speicherstelle $0001).

Die folgenden Beispiele sollen Ihnen die Arbeit mit dem Disk-Monitor verdeutlichen.

Achtung! Benutzen Sie unbedingt zum Üben eine Diskette, die Sie nicht mehr brauchen!

Weder wir noch der Verlag haften dafür, wenn Ihr Lieblingsprogramm oder die mühsam erstellte Adreßdatei unwiederbringlich dahin sind. Daß das sehr sehr schnell gehen kann, wissen wir aus eigener Erfahrung ...

Am besten machen Sie von einer Ihrer Disketten eine Kopie, die Sie zum Üben benutzen können.

Reparatur eines gelöschten Files

Sicher ist Ihnen das auch schon passiert: Sie wollen Ihr Programm mit Namen »Schrott« löschen, geben als Abkürzung »S:S*« ein und merken in dem Moment, in dem Sie »RETURN« drücken, daß auf der Diskette auch alle Versionen von »SMON« waren, außerdem auch noch »Superbase«, »Soccer« etc. Verzweifeln müssen Sie nur, wenn auch diese letzte SMON-Version mit dem Disk-Monitor dabei war. Ansonsten behalten Sie die Ruhe und verfahren Sie wie im folgenden beschrieben.

Laden Sie also jetzt SMON, legen Sie Ihre »Übungsdiskette« (!) ins Laufwerk und löschen Sie eins der ersten Programme mit dem üblichen Scratch-Kommando. Nun starten Sie SMON und drücken »Z«. Der Bildschirm ändert seine Farbe wie beschrieben und am Anfang der Zeile erscheint der »*«. Jetzt geben Sie ein:

R 12 00

Auf dem Bildschirm erscheint die erste Zeile der BAM, die bei jeder Diskette auf Track 18, Sektor 0 abgelegt ist. Die ersten beiden Byte enthalten »12 01« und geben damit den logisch nächsten Block an. In diesem Falle wäre das der erste Block des Directory. Wenn Sie mit »SHIFT« die Bildschirmausgabe fortsetzen, erkennen Sie etwa in der Mitte den Diskettennamen. Lassen Sie die Ausgabe durchlaufen, bis wieder der »*« erscheint. Nun geben Sie »R« ohne weitere Angaben ein. Damit erhalten Sie den Koppel-Block, also Track 18, Sektor 1, den ersten Directory-Block. (Natürlich hätten Sie auch gleich »R 12 01« eintippen können, aber wir wollen ja zeigen, wie die Befehle funktionieren.)

In diesem Block stehen die ersten acht Programme Ihrer Übungsdiskette, auch der Name des soeben gelöschten ist dabei.

Dateien einfach manipulieren

Trotzdem ist dieses Programm tatsächlich gelöscht und erscheint nicht mehr, wenn Sie sich das Directory anzeigen lassen. Vergleichen Sie den Eintrag des gelöschten Programms mit den anderen, fällt auf, daß 3 Byte vor Beginn des Namens bei allen anderen »82« steht (sofern es sich um Programmfiles handelt), bei dem gelöschten aber »00«. Die Reparatur ist nun denkbar einfach: Sie brauchen lediglich die »00« mit »82« zu überschreiben. Einen Haken hat die Sache allerdings noch. Beim SCRATCHEN sind die vom Programm belegten Blöcke in der BAM als frei gekennzeichnet worden und jeder neue Eintrag würde das als gelöscht gekennzeichnete File endgültig überschreiben. Um das zu verhindern, müssen Sie nach erfolgter Reparatur die Diskette validieren (von Basic aus mit Kommando: OPEN 1, 8, 15, "V"). Dabei wird die BAM neu erzeugt und korrigiert.

Schützen eines Files

Da wir gerade dabei sind, wollen wir unser repariertes gelöschtes File gleich ein für allemal gegen Löschen schützen. Diese Möglichkeit des Diskettenoperationssystems (DOS) ist zwar nicht im Handbuch beschrieben, funktioniert aber trotzdem ausgezeichnet. Laden Sie dazu nochmals die erste Seite des Directory mit

R 12 01

und ändern Sie die »82« vor dem Fileeintrag in »C2«. Geben Sie »W« ein, um die Änderung auf Diskette zu schreiben. Verlassen Sie nun SMON mit »X« und lassen Sie sich ein Directory anzeigen. Das geschützte File ist mit einem »>« gekennzeichnet. Versuchen Sie nun, dieses Programm mit dem Scratch-Kommando zu löschen. Es geht nicht! Zum »Entriegeln« brauchen Sie nur das »C2« wieder in »82« zu ändern. Der »>« im Directory verschwindet und das File ist nicht mehr geschützt.

Schützen einer Diskette

Wollen Sie eine ganze Diskette vor versehentlichem Löschen oder Formatieren schützen, gibt es die Möglichkeit, die Löschschutzkerbe abzukleben. Es geht jedoch auch anders.

Achtung! Die im folgenden beschriebene Prozedur läßt sich nicht ohne weiteres rückgängig machen, auch nicht mit dem Disk-Monitor!

Nehmen Sie also eine Diskette, die Sie anschließend »hart formatieren« können (also mit Eingabe einer ID). Starten Sie nun den Disk-Monitor und lesen Sie die BAM mit »R 12 00« ein. Das dritte Byte enthält »41«. Diese »41« ist ein Kennzeichen für das DOS der 1541- oder 4040-Floppy. Ändern Sie diese Byte durch Überschreiben in »45« und speichern Sie die Änderung mit »W« auf die Diskette zurück. Verlassen Sie nun SMON und versuchen Sie, etwas zu löschen. Ergebnis siehe oben. Versuchen Sie auch, die Diskette »weich«, also zum Beispiel mit OPEN 1,8,15,"N:TEST" zu formatieren.

Auch das ist jetzt nicht mehr möglich. Aber es kommt noch besser: Starten Sie noch einmal den Disk-Monitor und versuchen Sie, die Änderung durch Zurückschreiben der »41« an Stelle der »45« rückgängig zu machen. Auch das ist nicht mehr möglich, wir hatten Sie bereits gewarnt! Es bleibt lediglich die Möglichkeit, die Diskette »hart«, zum Beispiel mit OPEN 1,8,15,"N:TEST,TE" zu formatieren. Sollten Sie nun entgegen allen Warnungen doch Ihre Master-Diskette gegen Schreibzugriffe gesichert haben, verraten wir Ihnen ausnahmsweise, wie Sie den Eingriff trotzdem rückgängig machen können. Dazu überlisten wir das DOS des 1541-Laufwerkes, indem wir ihm vorgaukeln, es hätte eine Diskette im Normalformat vor sich. Wir verwenden den Memory-Write-Befehl, mit dem wir in die Speicherstelle 0101 (Zero-Page Adresse) des 1541-RAM einfach ein »A« schreiben. Der CHR$-Code des »A« ist 65, oder in hexadezimaler Schreibweise 41. Erinnern Sie sich? Dieser Wert stand ursprünglich im dritten Byte des Tracks 18, Sektor 0. Mit folgendem kleinen Programm umgehen wir einfach die DOS-Kennzeichnung und wir können die Diskette wieder normal beschreiben. Am sinnvollsten ist es, sofort den SMON zu starten, das vorher in 45 abgeänderte Byte wieder in 41 zu verwandeln und abzuspeichern. Die Diskette kann dann wieder zum Lesen und Schreiben verwendet werden. Hier nun das kleine Programm:

```
10 OPEN 1,8,15
20 PRINT#1, "M-W"CHR$(1)CHR$(1)CHR$(1)CHR$(65)
30 CLOSE1
```

Ändern des Diskettennamens oder der ID

Wir haben bereits oben gesehen, daß in Spur 18, Sektor 0 einer Diskette etwa in der Mitte der Diskettenname gespeichert wird. Dieser Name kann durch einfaches Überschreiben geändert werden; er darf bekanntlich bis zu 16 Zeichen enthalten. Hat Ihr neuer Name weniger Buchstaben als der alte, müssen Sie die Lücken mit »A0« und nicht mit »20« als Leerzeichen ausfüllen. Dies gilt vor allem, wenn Sie mit dieser Methode Filenamen ändern wollen. Das geht natürlich im Prinzip genauso wie eben beschrieben. Hinter dem Diskettennamen ist in Spur 18, Sektor 0 die ID abgelegt. Sie wird beim Formatieren vor jeden Sektor in einen sogenannten Header geschrieben und dient dem DOS zur Identifikation der Diskette. Zusätzlich wird sie noch in der BAM gespeichert, damit sie beim Laden eines Directory mit angezeigt werden kann. Nun ist es grundsätzlich nicht möglich, die ID im Header eines Sektors ohne Formatieren zu ändern, wohl aber die Eintragung in der BAM und damit die ID, die im Directory angezeigt wird. Genau wie beim Namen ist dies durch einfaches Überschreiben in der BAM möglich.

Ändern eines Filetyps

Wenn Sie einmal versucht haben, ein sequentielles File, etwa eine Datei, mit LOAD zu laden, werden Sie gemerkt haben, daß dies nicht möglich ist. Das DOS behauptet einfach, ein solches File existiere nicht und der Computer meldet »FILE NOT FOUND«. Viele Spiele zum Beispiel legen die »Hall of Fame« oder Highscore-Liste als sequentielle Datei ab. Mit dem Disk-Monitor ist es nun aber möglich, den Filetyp im Directory zu verändern. Erinnern Sie sich an die »82«, die im Directory vor jedem Filenamen steht. Bei sequentiellen Files steht dort »81«. Was zu tun ist, werden Sie sich denken können. Na klar, die »81« wird in »82« geändert, und schon ist die Datei ohne weiteres ladbar, natürlich wieder erst nach dem Zurückschreiben mit »W«.

Sinnvoll ist dies natürlich nur von SMON aus (mit Eingabe einer Ladeadresse). Mit »M« oder »K« können Sie dann die Datei ansehen und natürlich auch ändern. Vergessen Sie nicht, die geänderte Datei nach dem Zurückschreiben wieder in ein sequentielles File zu verwandeln. Verblüffen Sie Ihre Freunde doch mal mit einem auf diese Weise »errungenen« High-Score. Die Anerkennung Ihrer Umwelt ist Ihnen sicher!

Ändern der Startadresse eines Programms

Wir haben uns bisher auf Manipulationen in der BAM oder im Directory beschränkt. Wollen wir in einem Programm selbst Änderungen vornehmen, müssen wir etwas tiefer in die »Geheimnisse der Floppy« eindringen. So ist es bisweilen interessant, die Startadresse eines Maschinenprogramms zu kennen oder zu ändern. Dazu gehen wir folgendermaßen vor: Zunächst suchen wir mit »R 12 01« und eventuell weiteren Folgesektoren (12 04, 12 07...) den Fileeintrag im Directory. Die beiden Byte hinter der »82« direkt vor dem Programmnamen geben an, auf welcher Spur und in welchem Sektor das Programm startet. Wenn dort zum Beispiel »0A 04« steht, beginnt das Programm auf Spur 10, Sektor 4. Lesen Sie nun diesen Block mit »R 0A 04« ein. Die ersten beiden Bytes dieses Blocks zeigen auf den nächsten Block des Programms, die beiden nächsten Bytes enthalten die Startadresse in der üblichen Low-High-Byte-Reihenfolge. Zum Ändern der Startadresse überschreiben Sie die Bytes mit der neuen und speichern den Block mit »W« auf die Diskette zurück.

Die Zusammenarbeit mit SMON

Mit all diesen Beispielen sind die Möglichkeiten des Disk-Monitors noch lange nicht erschöpft. Sie sollten Ihnen als Anregung für eigene Experimente dienen. Üben Sie aber unbedingt so lange, bis Sie alle Kommandos aus dem »FF« (oder dezimal 255) beherrschen. Sie ersparen sich damit unnötigen Ärger und durchweinte Nächte. Besonders interessant ist es, von SMON aus auf den Puffer zuzugreifen und die SMON-Befehle auf den Puffer anzuwenden. Erwähnen möchte ich nur die Möglichkeit, Programme für das DOS direkt zu assemblieren und in einem bestimmten Sektor ablegen zu können, die »Find«-Routinen oder das »K«-Kommando für Textänderungen. Da der Puffer im RAM unter dem Basic liegt, muß Basic in solchen Fällen abgeschaltet werden. Ändern Sie dazu mit dem »M«-Befehl in Speicherstelle 0001 die »37« in »36«.

Haben Sie die Arbeit mit SMON beendet, können Sie mit »Z« in den Disk-Monitor schalten und den Pufferbereich mit »W« (Spur, Sektor) abspeichern.

Die Ausgabe von Diskettenfehlern

Beim Arbeiten mit dem Disk-Monitor werden sämtliche Fehler vom Laufwerk direkt, auch ohne Eingabe von »@«, ausgegeben, zum Beispiel »ILLEGAL TRACK OR SECTOR«, wenn Sie mit »R« einen Block lesen wollen, den es gar nicht gibt. Einen Fehler hat das Programm allerdings, den wir nicht verschweigen wollen. Der letzte Block eines Files enthält als Koppeladresse »00 FF«. Da es einen solchen Block nicht geben kann, »weiß« das DOS, daß es am Ende angelangt ist. Versuchen Sie aber, den nächsten Block (Spur 0, Sektor 255!!) mit »R« zu lesen, erscheint als Fehlermeldung nicht, wie es sein müßte, »ILLEGAL BLOCK OR SECTOR«, sondern »SYNTAX ERROR«. Das ist zwar eigentlich unerheblich, sollte aber erwähnt werden. Der Fehler liegt in der Routine, die unsere Zahleneingaben in das richtige Diskettenformat wandelt. Es fehlte einfach der Platz im Programm für eine »korrekte« Umwandlung, wir mußten uns mit einer »Sparroutine« behelfen.

Abschließend noch ein SMON-Trick, den wir einem aufmerksamen Leser verdanken. Für eine Directory-Ausgabe fehlte der Platz im SMON. Es geht aber hilfsweise so: Laden Sie das Directory zum Beispiel mit

L"$" 8000

an einen freien Speicherplatz. Mit »M« oder »K« können Sie jetzt das Directory »lesen«. Damit sind alle wichtigen Funk-

tionen für den Umgang mit der Diskette im SMON enthalten.

Zwei Erweiterungen haben wir Ihnen zu Beginn angekündigt, die SMON noch leistungsfähiger machen sollen.

SMON lüftet Geheimnisse

Dabei handelt es sich einmal um eine Erweiterung des Disassemblers, mit dem nun auch die »illegalen« Opcodes des 6502 disassembliert werden, zum anderen um neue Funktionen beim Diskmonitor, mit denen Sie in den Innereien Ihrer Floppy herumstöbern können. Nun ist der Speicherplatz bis auf 5 Byte ausgeschöpft, und die 4-KByte-Grenze soll auf keinen Fall überschritten werden. Wir haben daher andere Funktionen herausgenommen, und zwar für die Disassembler-Erweiterung den Diskmonitor und für die Diskmonitor-Erweiterung den Trace-Modus.

Beide Erweiterungen sind also nicht gleichzeitig einsetzbar; überhaupt ist es sinnvoll, eigene Versionen für spezielle Anwendungen zusammenzustellen, eine »normale«, eine Spezial-Disk-Version und eine für verschärftes Disassemblieren.

Beginnen wir mit dem letzten: Wie Sie wissen, erscheinen beim Disassemblieren immer drei Sternchen, wenn SMON auf ein Byte trifft, das keinen gültigen 6510-Opcode darstellt. Nun wissen Sie aber vielleicht auch, daß es über den offiziellen Befehlssatz hinaus noch einige Befehle gibt, die der Hersteller des Prozessors zwar nicht dokumentiert hat, die aber nichtsdestotrotz funktionieren und in einigen Programmen auch ausgenutzt werden. Es wäre natürlich schön, wenn SMON auch diese »illegalen« Opcodes anzeigen könnte. Unsere Erweiterung macht's möglich.

Wir haben Mnemonics für eine Reihe dieser Befehle eingesetzt und lassen diese von SMON mit einem vorangestellten »*« ausgeben. Übrig bleiben noch zehn Befehle, deren Wirkung aber so komplex ist, daß sie sich beim besten Willen nicht mit einem Mnemonic abkürzen lassen. Sie fallen auch aus der Logik der Prozessorstruktur heraus. Im einzelnen handelt es sich um die Opcodes 0B, 2B, 4B, 6B, 8B, 9C, 9E, AB, CB und EB. Bei diesen Befehlen haben wir keine gemeinsame Struktur entdecken können. Die neuen Mnemonics haben folgende Bedeutung:

LAX	Load Akku and X entspricht LDA und LDX.
DCP	Decrement and ComPare entspricht DEC und CMP.
ISC	Increment and SubtraCt entspricht INC und SBC.
RLA	Rotate Left AND Akku entspricht ROL und AND.
RRA	Rotate Right and Add with carry entspricht ROR und ADC.
SLO	Shift Left OR Akku entspricht ASL und ORA.
SRE	Shift Right and EOR Akku entspricht LSR und EOR.
SAX	Store Akku AND X führt eine UND-Verknüpfung zwischen Akku und X-Register durch und speichert das Ergebnis in der angegebenen Adresse ab.
CRA	CRAsh führt zum »Absturz« des Prozessors.
NOP	NO Operation entspricht dem bekannten NOP, jedoch kann dieser Befehl auch 2 oder 3 Byte lang sein. Dies wird durch die angegebene Adresse deutlich, die in diesem Fall natürlich keinerlei Bedeutung hat.

Über den Sinn dieser Befehle läßt sich sicher streiten; allerdings kommen sie bisweilen in Programmen vor, meist um das Lesen dieser Programme unmöglich zu machen, also als Programmschutz. Von der Verwendung dieser Befehle in eigenen Programmen raten wir auf jeden Fall ab. Erstens wird kein Hersteller garantieren, daß die »illegalen« tatsächlich mit jedem 6510-Prozessor funktionieren, zweitens gibt es keine Funktion, die nicht auch mit den »normalen« Befehlen ebensogut erreicht werden könnte. Und als Programmschutz taugen die »illegalen« spätestens mit der Veröffentlichung dieses Artikels ja auch nichts mehr. Aus diesem Grund haben wir bewußt auf eine Erweiterung des Assemblers in dieser Richtung verzichtet. Sie können also keine normalen Opcodes durch Überschreiben in »illegale« ändern, wohl aber umgekehrt. Es bleibt lediglich die Eingabe als Einzelbyte, was aber hoffentlich zu umständlich ist.

Komfortabler Disketten-Monitor für SMON

Jetzt folgt unser zweiter Leckerbissen in Form eines kleinen aber ungemein wertvollen Zusatzprogrammes für den SMON. Es handelt sich dabei um eine Erweiterung des Disketten-Monitors, mit dem jeder auf einen Schlag die Arbeit von Stunden zunichte machen kann. Geben Sie das Programm wie beschrieben ein, starten Sie SMON wie gewohnt und springen mit »Z« in den Disketten-Monitor. Von hier aus erreichen Sie mit »F« (wie Floppy) die neuen Befehle. Wir haben absichtlich diesen umständlichen Weg gewählt, denn Fehler in diesem Modus wirken noch dramatischer als sonst. Mit diesem Werkzeug haben Sie unmittelbaren Zugriff auf die Eingeweide der Floppy. Jetzt können Sie die folgenden Befehle mit einer Übungsdiskette (!!!) in aller Ruhe durcharbeiten.

M Memory-Dump des Disketten-Monitors

Beispiel: M (ohne weitere Eingabe) listet den Bereich des Floppy-RAM von $0000-$00FF. (Es erscheint zunächst die erste Zeile, weitere Ausgabe mit der SPACE-Taste.)

In diesem Bereich befinden sich unter anderem die Jobspeicher ($00-$04) für die fünf Puffer 0 bis 4 sowie die wichtigsten Variablen des DOS.

M 07 Memory-Dump ab $0700

Die BAM der Diskette wird nach dem Initialisieren in Puffer 4 ($0700 im Floppy-RAM) eingelesen. Schauen Sie sich also mit »M 07« die aktuelle BAM an. Sie könnten jetzt durch einfaches Überschreiben den Inhalt der BAM ändern. (Der Doppelpunkt vor der Zeile wirkt als »hidden command«). Dann schauen Sie sich Ihre Änderung mit »M 07« wieder an. Sie sehen, daß inzwischen der Inhalt des Floppy-RAM geändert wurde. Wenn Sie nun den Jobcode »90« (= Schreibbefehl an den Floppy-Controller) in Speicherstelle $04 bringen, würde die geänderte (falsche!) BAM auf Diskette zurückgeschrieben werden!! Es gibt also genug Möglichkeiten, wie oben angedeutet, die Disketten zu »versauen«.

Für das Ausprobieren noch einige wichtige Speicherstellen und Jobcodes:

$80	Lesen
$90	Schreiben
$C0	»Anschlagen« des Kopfes
$D0	Maschinenprogramme im Puffer ausführen
$E0	Programm im Puffer ausführen mit Hochfahren des Laufwerks

Speicherstellen im Floppy-RAM:

$06/$07	ist Spur- und Sektornummer für den Befehl in Puffer 0
$08/$09	für Puffer 1
$0A/$0B	für Puffer 2

$0C/$0D für Puffer 3
$0E/$0F für Puffer 4

Jedem Puffer sind zwei Speicherstellen zugeordnet, eine für den Jobcode ($0000 bis $0004) und eine für Spur und Sektor. Wenn Sie also in Puffer 0 (in $0300 gelegen) einen bestimmten Block einlesen wollen, geben Sie folgende Befehle ein:

»M« liest die Zeropage der Floppy ein — so sehen dann zum Beispiel die ersten Zeilen aus:

```
:0000 01 01 01 FF 03 04 01 34
:0008 23 02 04 50 01 03 0A 11
```

Gehen Sie mit dem Cursor in die erste Zeile und schreiben Sie »80« in die erste Speicherstelle (anstelle der ersten 01). In Speicherstelle $06/$07 (die letzten beiden in der ersten Reihe) die Spur- und die Sektornummer, die gelesen werden soll, zum Beispiel 12 01. Sie sehen dann

```
:0000 80 01 01 FF 03 04 12 01
:0008 unverändert
```

Drücken Sie die RETURN-Taste, Mit »M 03« kann jetzt der eingelesene Block (hier der erste Directory-Block) angesehen werden. Änderungen können durch einfaches Überschreiben vorgenommen werden. Dauerhaft wird Ihre Änderung erst durch Zurückschreiben (nach Spur $12 und Sektor $01) mit dem Jobcode »90« in der ersten Speicherstelle. Nach Änderung der beiden für Puffer 0 zuständigen Adressen ($06/$07) auch an jede beliebige andere Stelle. Das ist wörtlich zu nehmen, denn wir befinden uns hier »unterhalb« der Controllerebene, die unter anderem für die Prüfung auf Einhaltung der zulässigen Spur und Sektorgrenzen verantwortlich ist. Es erfolgt also keine Fehlermeldung, wenn Sie versuchen sollten, mit Ihrer Floppy bis in die des Nachbarn zu schreiben (zum Beispiel mit der Spur 152).

Entsprechende Lese- und Schreibübungen können mit den anderen Puffern durchgeführt werden. Denken Sie daran, erst ist die Spur- beziehungsweise Sektornummer für den entsprechenden Puffer (in der zweiten Zeile!) einzugeben, bevor Sie in Zeile 1 den Jobcode mit einem »RETURN« übergeben, denn mit Druck auf die RETURN-Taste wird Ihr Befehl ausgeführt. Und noch eins: Quälen Sie bitte dabei Ihren Schreibkopf nicht mehr als unbedingt erforderlich, sonst könnte er sich mechanisch verklemmen und nur noch mit einem Eingriff in die Floppymechanik wieder »befreit« werden.

Falls Sie die Ausgaben 1/85 (Seite 151) und 3/85 (Seite 103 bis 135) der 64'er besitzen, können Sie sich dort über andere Speicherstellen der Floppy und die weitere Anwendung der Jobcodes informieren.

Der Befehl @ ohne weitere Angaben fragt den Fehlerkanal ab, ansonsten dient er zur Befehlsübermittlung an die Floppy.

Beispiel:	@	Fehlerkanal
	@I	Initialisierungsbefehl oder
	@S:name	Befehl zum Scratchen und so weiter.

Bedingt durch die verschiedenen Versionen, springt dieser Befehl manchmal in den »normalen« Disketten-Monitor zurück, erkennbar an dem »*« am Zeilenanfang. Sie müssen dann wieder ein »F« eingeben.

Mit X gelangt man wieder in den Disketten-Monitor.

Zum Abschluß ein sehr hilfreicher Befehl namens »V«, der es erlaubt, Speicherbereiche aus dem Computer in den Laufwerkspuffer zu verschieben. Folgende einfache Syntax gilt dabei: **V von nach**

Um zum Beispiel ein Maschinenprogramm von $6000 in den Puffer 1 zu bekommen, geben Sie folgendes ein:
V 6000 0400

Dabei wird immer eine ganze Seite, also 256 Byte, übertragen. Was das Programm dort soll, fragen Sie? Führen Sie es doch einfach aus (Jobcode $D0 in Speicherstelle $01 schreiben); oder schreiben Sie es mit dem Jobcode »90« in einen beliebigen Sektor der Diskette.

Wenn Sie dann Ihre Floppy so richtig durcheinandergebracht haben und nichts läuft mehr, brauchen Sie nicht zu verzweifeln. Außer einem eventuell festhängenden Lesekopf passiert der Floppy nichts, nur Ihren Disketten.

Hinweise zum Abtippen

Tippen Sie die beiden Erweiterungsprogramme (Listing 2 und 3 mit dem MSE-Programm ab und speichern Sie die fertigen Programme.

Laden und starten Sie dann Ihren SMON $C000. Geben Sie ein: L"NDISASS"

Damit werden die neuen Befehle automatisch über den bisherigen Disketten-Monitor geladen. Sie müssen nun aber noch aktiviert werden. Geben Sie dazu G CF0D ein.

SMON meldet sich sofort mit seiner Registeranzeige wieder. Sie sollten nun diese Version unbedingt speichern, zum Beispiel mit S"SMON NDISASS" C000 CF3D

Wenn Sie nun das Programm »ILLEGAL-CODE« (Listing 4) laden und mit D 4000 disassemblieren, sehen Sie die »illegalen« Opcodes schön geordnet nacheinander.

Um die neuen Befehle des Disketten-Monitors in SMON einzubinden, gehen Sie ganz ähnlich vor. Nach dem Abtippen und Speichern des Programms »FLOPPYMON« muß natürlich SMON C000 geladen und gestartet werden. Anschließend geben Sie ein: L"FLOPPYMON" und aktivieren es mit G CDD8.

Zum Speichern geben Sie S"SMON-FLOPPY" COOO CFFF ein.

SMON erweitern

Die zum Schluß vorgestellte Erweiterung stellt elf weitere Befehle für den SMON zur Verfügung. So läßt sich der Monitor zum Beispiel frei im Speicher verschieben und Sprites oder Zeichensätze können sehr einfach erstellt und geändert werden.

Um die Befehlserweiterung zu initialisieren, geht man folgendermaßen vor:

1. SMON absolut laden und NEW eingeben.
2. Den Basic-Lader (Listing 5) eintippen und speichern.
3. Nach dem Start des Laders die Startadresse (dezimal) Ihrer
SMON-Version eingeben: zum Beispiel 49152 (= $C000).
4. Den erweiterten SMON zum Beispiel mit "SMONEX" Startadresse Endadresse speichern.Die neuen Routinen werden, genau wie die meisten bereits vorhandenen, durch einen Buchstaben, zum Teil gefolgt von Adressenangaben, aufgerufen. Bei den ersten drei Ausgabebefehlen kann der Speicherinhalt durch Überschreiben der Zeile geändert werden.

Z 4000 (4100) (Zeichendaten)
gibt den Speicherinhalt von $4000 (bis $40FF) folgendermaßen aus: Jeweils ein Byte pro Zeile wird in 8-Bit-Form dargestellt. Dabei ist ein »*« ein gesetztes, ein ».« dagegen ein nicht gesetztes Bit. Die beiden Zeichen sind willkürlich gewählt und können durch Überschreiben der Speicherzellen $xE65, $xE2D (Bit = 1) und $xE69, $xE30 (Bit = 0) in den Bildschirm-Code (!) der gewünschten Zeichen geändert werden.

Die Anwendung dieses Befehls liegt beispielsweise in der gezielten und anschaulichen Beeinflussung bestimmter Steuerbits in VIC, CIA etc. Andererseits lassen sich — besonders in Verbindung mit dem Kommando »Q« — Zeichendaten leicht modifizieren.

H 4000 (4100)
entspricht dem Befehl »Z« mit dem Unterschied, daß jeweils drei Byte pro Zeile ausgegeben werden. Das entspricht dem Format für Spritedaten. Auf diese Weise steht mit dem erweiterten SMON ein kleiner »Sprite-Editor« zur Verfügung.

N 4000 (4100) (Normaldarstellung)
interpretiert den Speicherinhalt von $4000 (bis $40FF) als Bildschirm-Code und gibt 32 Zeichen pro Zeile aus.

U 4000 (4100) (Übersicht)
Wie »N«, jedoch werden in einer Zeile 40 Zeichen dargestellt. Änderungen sind nur mit »N« möglich. Dieser Befehl dient hauptsächlich dazu, im Speicher abgelegte Bildschirminformationen so auszugeben, wie sie tatsächlich im 40-Zeichen/Zeile-Format aussehen würden. Dieser Befehl ist recht nützlich, um professionelle Videospiele zu analysieren, da hier Spielszenen oft im Bildschirm-Code gespeichert sind.

E 4000 (4100) (Erase)
ist der bereits im 64'er, Ausgabe 2/85 vorgeschlagene Erase-Befehl zum Füllen des Speicherbereiches von $4000 bis $40FF mit $00.

Y 40
kopiert die vorhandene SMON-Version in nur drei Sekunden nach $4000 bis $4FFF und nimmt dabei alle notwendigen Anpassungen vor. Die ursprüngliche Speicherversion des Monitors bleibt unverändert. Mit »G 4000« kann man in den neuen SMON springen. Von dem Byte-Wert, der übergeben werden muß, wird nur das obere Nibble ($4) gewertet, so daß sich theoretisch 16 SMON-Versionen im Speicher unterbringen lassen, wobei natürlich nicht alle Möglichkeiten sinnvoll sind. Auf diese Weise läßt sich stets die

Befehlsübersicht zum SMON

Alle Eingaben erfolgen in der hexadezimalen Schreibweise. In Klammern angegebene Adreßeingaben können entfallen. SMON benutzt dann sinnvolle, vorgegebene Werte.

Bei allen Ausgabe-Befehlen ist gleichzeitig die Ausgabe auf einem Drucker möglich. Dazu werden die Befehle geSHIFTet eingegeben.

A	**4000** (Assembler) symbolischer Assembler (Verarbeitung von Label möglich) Startadresse $4000
B	**4000 4200** (Basic-Data) erzeugt Basic-DATA-Zeilen aus Maschinenprogramm im Bereich von $4000 bis $41FF
C	**4010 4200 4013 4000 4200** (Convert) in ein Programm, das von $4000 bis $4200 im Speicher steht, soll bei 4010 ein 3-Byte-Befehl eingefügt werden. Dazu wird das Programm ab $4010 bis 4200 auf die neue Adresse $4013 verschoben. Alle absoluten Adressen, die innerhalb des Programmbereichs ($4000 bis $4200) stehen, werden umgerechnet, so daß die Sprungziele stimmen.
D	**4000 (4100)** (Disassembler) disassembliert den Bereich von $4000 (bis $4100) mit Ausgabe der Hex-Werte. Änderungen sind durch Überschreiben der Befehle möglich.
F	(Find) findet Zeichenketten (F), absolute Adressen (FA), relative Sprünge (FR), Tabellen (FT), Zeropageadressen (FZ) und Immediate-Befehle (FI)
G	**(4000)** (Go) startet ein Maschinenprogramm, das bei $4000 im Speicher beginnt
I	**01** (I/O-Gerät) stellt die Gerätenummer für Floppy (08 oder 09) oder Datasette (01) ein
K	**A000 (A500)** (Kontrolle) zum schnellen Durchsuchen des Bereichs von $A000 (bis $A500) nach ASCII-Zeichen (32 Byte pro Zeile). Änderungen sind durch Überschreiben der ASCII-Zeichen möglich.
L	**(4000)** (Load) lädt ein Maschinenprogramm an die richtige oder eine angegebene Adresse ($4000)
M	**4000 (4400)** (Memory-Dump) gibt den Inhalt des Speichers von $4000 (bis $43FF) in Hex-Byte und ASCII-Code aus. Änderungen sind durch Überschreiben der Hex-Zahlen möglich.
O	**4000 4500 AA** (Occupy) füllt den Speicherbereich von $4000 bis $4500 mit vorgegebenem Byte ($AA) aus
P	**05** (Printer) setzt Geräteadresse 5 für Drucker
R	(Register) zeigt die Registerinhalte und Flags an. Änderungen sind durch Überschreiben möglich.
S	**"Test" 4000 5000** (Save) speichert ein Programm von $4000 bis $4FFF unter dem Namen »Test« ab
TW	**(4000)** (Trace Walk) führt auf Tastendruck den jeweils nächsten Maschinenbefehl aus und zeigt die Registerinhalte an. Subroutine können in Echtzeit durchlaufen werden (»J«). Wird keine Startadresse eingegeben, beginnt »TW« bei der letzten mit »R« angezeigten Adresse.
TB	**4010 05** (Trace Break) setzt einen Haltepunkt für den Schnellschrittmodus bei $4010. Der Schnellschrittmodus wird unterbrochen, nachdem $4010 zum fünften Mal erreicht worden ist.
TQ	**4000** (Trace quick) Schnellschrittmodus, springt beim Erreichen eines Haltepunktes in den Einzelschrittmodus
TS	**4000 4020** (Trace stop) arbeitet ein Programm ab $4000 in Echtzeit ab und springt beim Erreichen von $4020 in die Registeranzeige. Von dort aus kann (nach eventueller Änderung der Register) mit »G« oder »TW« fortgefahren werden. »TS« arbeitet nur im RAM-Speicher.
V	**6000 6200 4000 4100 4200** (Verschieben) ändert in einem Programm von $4100 bis $41FF alle absoluten Adressen, die sich auf den Bereich von $6000 bis $6200 beziehen, auf einen neuen Bereich, der bei $4000 beginnt.
W	**4000 4300 5000** (Write) verschiebt den Speicherinhalt von $4000 bis $42FF nach $5000 ohne Umrechnung der Adressen (zum Beispiel Tabellen)

erforderliche Speicherversion herstellen, ohne daß langwierige Änderungen notwendig sind.

Q 2000
kopiert den Zeichensatz aus dem ROM von $D000 bis $DFFF in das RAM nach $2000. Dort kann er mit dem Befehl »Z« nach Belieben geändert werden. Möchte man zum Beispiel das Zeichen »A« in ein »Ä« umdefinieren, so ist der Zeichensatz mit »Q 2000« ins RAM zu kopieren. Anschließend kann mit »Z 2000 2015« der Bereich in binärer Form auf dem Bildschirm ausgegeben werden, in dem auch das Zeichen »A« steht. Dieses kann nun in ein »Ä« geändert werden, indem man mit dem Cursor an die zu ändernde Stelle fährt und für einen Punkt, der gesetzt werden soll, ein »*« und für einen Punkt der nicht gesetzt werden soll ein ».« setzt. So, jetzt ist der Zeichensatz umdefiniert, aber noch nicht aktiviert. Als nächstes muß dem Videocontroller die Startadresse des neuen Zeichensatzes mitgeteilt werden. Dazu ist die Adresse $D018, in der eine hexadezimale 15 steht, durch eine hexadezimale 18 zu ersetzen.

X (Exit)
springt aus dem Monitor-Programm ins Basic zurück

**49152**
Dezimalzahl umrechnen

$ **002B**
4stellige Hex-Zahl umrechnen

% **01101010**
8stellige Binärzahl umrechnen

? **0344 + 5234**
Addition oder Subtraktion zweier 4stelliger Hex-Zahlen

= **4000 5000** (Vergleich)
vergleicht den Speicherinhalt ab $4000 mit dem ab $5000

Z (Diskmonitor)
ruft den Diskmonitor auf. Dieser verfügt über folgende Befehle:

R **(12 01)** (Read)
liest Track $12, Sektor $01 von der Diskette in einen Puffer im Speicher. Fehlt die Angabe von Track und Sektor, wird der logisch (!) nächste Sektor gelesen.

W **(12 01)** (Write)
schreibt den Puffer im Speicher nach Track $12, Sektor $01 auf die Diskette. Ohne Angabe von Track und Sektor werden die letzten Eingaben von »R« benutzt.

M (Memory-Dump)
zeigt den Pufferinhalt als Hexdump (wie normales »M«). Weitere Ausgabe mit CBM-Taste, Abbruch mit STOP. Werte können durch Überschreiben geändert werden.

X (Exit)
springt in SMON zurück

F (weitere Disketten-Befehle initialisieren)
sind die Befehle initialisiert, gilt:
M (07)
Memory-Dump (Floppy-RAM/ROM)
V 6000 0400
Verschieben eines 256-Byte-Blocks von $6000 in den Laufwerkspuffer 1 beziehungsweise in das Floppy-RAM
@
normale Disketten-Befehle senden
X
zurück zum normalen Disketten-Monitor

J (Wiederholung)
bringt den letzten Ausgabebefehl (K, D, M, Z, H, N, U) auf den Bildschirm zurück. Mit RETURN wird der letzte Befehl noch einmal ausgeführt.

Zum Schluß noch ein Tip:

DATA-Zeilen in Hex-Byte-Darstellung sind wegen ihrer konstanten Länge (immer zwei Ziffern pro Wert!) übersichtlicher als solche mit dezimalen Zahlen. Da für die Ausgabe von Hex-Werten bereits alle Routinen im SMON integriert sind, kann der »B«-Befehl (Basic-DATA-Zeilen erzeugen) durch Verändern eines einzigen Sprungbefehles dahingehend manipuliert werden, daß der Speicherinhalt künftig in Form von Hex-Byte ausgegeben wird:

Disassemblieren Sie dazu den Byte-Ausgabebefehl mit »D x99F« und ersetzen »JSR BDD1« durch »JSR x32A«. Für das »x« muß der 4-KByte-Block, in dem die zu ändernde SMON-Version steht, eingesetzt werden. Liegt Ihre SMON-Version bei $C000, so ersetzen Sie das »x« durch ein »C«.

Die Gesamtlänge der DATA-Zeile kann außerdem durch Verändern der Speicherzelle $x9AE variiert werden. Bei dem Wert $1C werden zum Beispiel genau acht Hex-Byte pro Zeile ausgegeben. Das Assembler-Listing zu dieser Erweiterung zeigt Listing 6.

(Dietrich Weineck/Mark Richters/sk)

SMON-Speicherstellen

Folgende Zeropage-Adressen werden benutzt:

FLAG	$AA	Universalflag
ADRCODE	$AB	Adressierungscode für Assembler/Disassembler
COMMAND	$AC	SMON-Befehlscode
BEFCODE	$AD	Befehlscode Ass./Disass.
LOPER	$AE	Low-Operand für Ass./Disass.
HOPER	$AF	High-Operand für Ass./Disass.
BEFLEN	$B6	Befehlslänge Ass./Disass.
PCL	$FB	SMON-Programmcounter Low-Byte
PCH	$FC	SMON-Programmcounter High-Byte

Außerhalb der Zeropage benutzt SMON die Bereiche:

PCHSAVE	$02A8	
PCLSAVE	$02A9	
SRSAVE	$02AA	
AKSAVE	$02AB	dienen der Zwischenspeicherung
XRSAVE	$02AC	der angegebenen Register
YRSAVE	$02AD	
SPSAVE	$02AE	
PRINTER	$02AF	Printernummer
IO.NR	$02B0	Devicenummer
MEM	$02B1	Buffer bis $02B8
TRACEBUF	$02B8 bis $02BF	Buffer für Trace-Modus

Dann folgen die von Diskmonitor benötigten Adressen:

SAVEX	$02C1	Zwischenspeicherung der X- und Y-Register
TMPTRCK	$02C2	
TMPSECTO	$02C3	Zwischenspeicher für Track und Sektor
DCMDST	$02D0	Diskkommandostring
TRACK	$02D8	
SECTO	$02DB	Track und Sektornummer
BUFFER	$033C bis $03FC	Buffer für Label, nur für Assembler

Einsprungadressen von SMON-Routinen

; (TICK)	$CADB	B (BASICDATA)	$C96C	P (SETPRINTER)	$C83D
# (BEFDEC)	$C92E	C (CONVERT)	$CA3D	R (REGISTER)	$C386
$ (BEFHEX)	$C908	D (DISASS.)	$C55D	S (LOADSAVE)	$C84E
% (BEFBIN)	$C91C	F (FIND)	$CB11	T (TRACE)	$CBF1
, (KOMMA)	$C6FC	G (GO)	$C3E3	V (VERSCHIEB)	$CA43
: (COLON)	$C41D	I (IO.SET)	$C844	W (WRITE)	$C9D3
; (SEMIS)	$C3B6	K (KONTROLLE)	$CAB7	X (EXIT)	$C36E
= (COMP)	$CAF5	L (LOADSAVE)	$C84E	Z (DMON)	$CE09
? (ADDSUB)	$C89A	M (MEMDUMP)	$C3F9		
A (ASSMBLER)	$C6D1	O (OCUPPY)	$C9C1		

```
Name : smon c000              c000 cffc
---------------------------------------
c000 : a9 14 8d 16 03 a9 c2 8d  7d
c008 : 17 03 00 27 23 24 25 2c  c6
c010 : 3a 3b 3d 3f 41 42 43 44  db
c018 : 46 47 49 4b 4c 4d 4f 50  cb
c020 : 52 53 54 56 57 58 5a 00  9d
c028 : 00 00 00 da ca 2d c9 07  cf
c030 : c9 1b c9 fb c6 1c c4 b5  44
c038 : c3 f4 ca 99 c8 d0 c6 6b  60
c040 : c9 3c ca 5c c5 10 cb e2  37
c048 : c3 43 c8 b6 ca 4d c8 f8  e2
c050 : c3 c0 c9 3c c8 85 c3 4d  d0
c058 : c8 f0 cb 42 ca d2 c9 6d  19
c060 : c3 08 ce 00 00 00 00 00  db
c068 : 00 00 00 ff ff 01 00 41  f3
c070 : 5a 49 52 54 80 20 40 10  b8
c078 : 00 02 01 01 02 00 91 91  63
c080 : 0d 53 d9 31 37 32 0d 00  0d
c088 : 7d 4c 7d c9 0d 0d 20 20  be
c090 : 50 43 20 20 53 52 20 41  59
c098 : 43 20 58 52 20 59 52 20  a2
c0a0 : 53 50 20 20 4e 56 2d 42  f8
c0a8 : 44 49 5a 43 00 02 04 01  b2
c0b0 : 2c 00 2c 59 29 58 9d 1f  1d
c0b8 : ff 1c 1c 1f 1f 1f 1c df  cb
c0c0 : 1c 1f df ff ff 03 1f 80  f9
c0c8 : 09 20 0c 04 10 01 11 14  da
c0d0 : 96 1c 19 94 be 6c 03 13  cf
c0d8 : 01 02 02 03 03 02 02 02  08
c0e0 : 02 02 02 03 03 02 03 03  17
c0e8 : 03 02 00 40 40 80 80 20  3f
c0f0 : 10 25 26 21 22 81 82 21  bb
c0f8 : 82 84 08 08 e7 e7 e7 e7  ed
c100 : e3 e3 e3 e3 e3 e3 e3 e3  ff
c108 : e3 e3 e7 a7 e7 e7 f3 f3  41
c110 : f7 df 26 46 06 66 41 81  e5
c118 : e1 01 a0 a2 a1 c1 21 61  66
c120 : 84 86 e6 c6 e0 c0 24 4c  b7
c128 : 20 90 b0 f0 30 d0 10 50  45
c130 : 70 78 00 18 d8 58 b8 ca  a8
c138 : 88 e8 c8 ea 48 08 68 28  7a
c140 : 40 60 aa a8 ba 8a 9a 98  0c
c148 : 38 f8 89 9c 9e b2 2a 4a  af
c150 : 0a 6a 4f 23 93 b3 f3 33  d5
c158 : d3 13 53 73 52 4c 41 52  29
c160 : 45 53 53 4f 4c 4c 4c 43  ec
c168 : 41 41 53 53 49 44 43 43  d3
c170 : 42 4a 4a 42 42 42 42 42  76
c178 : 42 42 42 53 42 43 43 43  a8
c180 : 43 44 44 49 49 4e 50 50  09
c188 : 50 50 52 52 54 54 54 54  c1
c190 : 54 54 53 53 4f 53 53 4f  c9
c198 : 4f 54 42 52 44 44 44 4d  fe
c1a0 : 4e 44 54 54 4e 45 50 50  a1
c1a8 : 49 4d 53 43 43 45 4d 4e  05
c1b0 : 50 56 56 45 52 4c 4c 4c  bb
c1b8 : 4c 45 45 4e 4e 4f 48 48  d3
c1c0 : 4c 4c 54 54 41 41 53 58  ee
c1c8 : 58 59 45 45 4c 52 4c 52  f4
c1d0 : 52 41 43 41 59 58 41 50  ba
c1d8 : 44 43 59 58 43 43 58 59  82
c1e0 : 54 50 52 43 53 51 49 45  c9
c1e8 : 4c 43 53 49 4b 43 44 49  46
c1f0 : 56 58 59 58 59 50 41 50  91
c1f8 : 41 50 49 53 58 59 58 41  52
c200 : 53 41 43 44 08 84 81 22  3c
c208 : 21 26 20 80 03 20 1c 14  1e
c210 : 14 10 04 0c d8 a9 08 8d  c5
c218 : b0 02 a9 04 8d af 02 a9  66
c220 : 06 8d 20 d0 8d 21 d0 a9  87
c228 : 03 8d 86 02 a2 05 68 9d  03
c230 : a8 02 ca 10 f9 ad a9 02  46
c238 : d0 03 ce a8 02 ce a9 02  94
c240 : ba 8e ae 02 a9 52 4c ff  8c
c248 : c2 20 c2 c2 f0 0b 20 7e  08
c250 : c2 8d a9 02 a5 fc 8d a8  4d
c258 : 02 60 a2 a4 20 80 c2 20  19
c260 : 80 c2 d0 1c 20 7e c2 a9  4d
c268 : fe 85 fd a9 ff 85 fe 20  46
c270 : c2 c2 d0 0c 8d 77 02 e6  b3
c278 : c6 60 20 7e c2 2c a2 fb  56
c280 : 20 8d c2 95 01 20 9a c2  cb
c288 : 95 00 e8 e8 60 20 ca c2  2c
c290 : c9 20 f0 f9 c9 2c f0 f5  92
c298 : d0 03 20 ca c2 20 af c2  bd
c2a0 : 0a 0a 0a 0a 85 b4 20 ca  87
c2a8 : c2 20 af c2 05 b4 60 c9  ca
c2b0 : 3a 90 02 69 08 29 0f 60  a7
c2b8 : 20 ca c2 c9 20 f0 f9 c6  26
c2c0 : d3 60 20 cf ff c6 d3 c9  de
c2c8 : 0d 60 20 cf ff c9 0d d0  2b
c2d0 : f8 a9 3f 20 d2 ff ae ae  b6
c2d8 : 02 9a a2 00 86 c6 20 51  92
c2e0 : c3 a1 d1 c9 27 f0 11 c9  f3
c2e8 : 3a f0 0d c9 3b f0 09 c9  0a
c2f0 : 2c f0 05 a9 2e 20 d2 ff  3a
c2f8 : 20 ca c2 c9 2e f0 f9 85  c4
c300 : ac 29 7f a2 20 dd 0a c0  10
c308 : f0 05 ca d0 f8 f0 c2 20  aa
c310 : 15 c3 4c d6 c2 8a 0a aa  f3
c318 : e8 bd 29 c0 48 ca bd 29  65
c320 : c0 48 60 a5 fc 20 2a c3  d2
c328 : a5 fb 48 4a 4a 4a 4a 20  87
c330 : 35 c3 68 29 0f c9 0a 90  0f
c338 : 02 69 06 69 30 4c d2 ff  4e
c340 : a9 0d 20 d2 ff 8a 4c d2  fd
c348 : ff 20 4c c3 a9 20 4c d2  55
c350 : ff a9 0d 4c d2 ff 85 bb  ab
c358 : 84 bc a0 00 b1 bb f0 06  2b
c360 : 20 d2 ff c8 d0 f6 60 e6  16
c368 : fb d0 02 e6 fc 60 a9 0e  be
c370 : 8d 86 02 8d 20 d0 a9 06  ae
c378 : 8d 21 d0 a9 37 85 01 ae  00
c380 : ae 02 9a 4c 74 a4 a0 c0  d0
c388 : a9 8c 20 56 c3 a2 3b 20  c9
c390 : 40 c3 ad a8 02 85 fc ad  ce
c398 : a9 02 85 fb 20 23 c3 20  8e
c3a0 : 4c c3 a2 fb bd af 01 20  93
c3a8 : 2a c3 20 4c c3 e8 d0 f4  f6
c3b0 : ad aa 02 4c d0 c3 20 4e  05
c3b8 : c2 a2 fb 20 ca c2 20 9a  47
c3c0 : c2 9d af 01 e8 d0 f4 20  86
c3c8 : 4c c3 bd aa 02 4c d0 c3  08
c3d0 : 85 aa a9 20 a0 09 20 d2  91
c3d8 : ff 06 aa a9 30 69 00 88  19
c3e0 : d0 f4 60 20 49 c2 ae ae  09
c3e8 : 02 9a a2 fa bd ae 01 48  25
c3f0 : e8 d0 f9 68 a8 68 aa 68  15
c3f8 : 40 20 64 c2 a2 3a 20 40  b7
c400 : c3 20 23 c3 a0 20 a2 00  aa
c408 : 20 4c c3 a1 fb 20 2a c3  64
c410 : a1 fb 20 39 c4 d0 f1 20  b9
c418 : 5d c4 90 e0 60 20 7e c2  9e
c420 : a0 20 a2 00 20 ca c2 20  1d
c428 : 9a c2 81 fb c1 fb f0 03  c9
c430 : 4c d1 c2 20 39 c4 d0 ec  f0
c438 : 60 c9 20 90 0c c9 60 90  49
c440 : 0a c9 c0 90 04 c9 db 90  90
c448 : 04 a9 2e 29 3f 29 7f 91  30
c450 : d1 ad 86 02 91 f3 20 67  e2
c458 : c3 c8 c0 28 60 20 6f c4  03
c460 : 4c 66 c4 20 67 c3 a5 fb  38
c468 : c5 fd a5 fc e5 fe 60 20  4d
c470 : 94 c4 20 86 c4 f0 0e 20  8b
c478 : 86 c4 f0 fb c9 20 d0 05  07
c480 : 8d 77 02 e6 c6 60 20 e4  e0
c488 : ff 48 20 e1 ff f0 02 68  50
c490 : 60 4c d6 c2 a0 28 24 ac  59
c498 : 10 f6 84 c8 84 d0 a9 ff  d3
c4a0 : 20 c3 ff a9 ff 85 b8 85  f1
c4a8 : b9 ad af 02 85 ba 20 c0  94
c4b0 : ff a2 00 86 d3 ca 20 c9  79
c4b8 : ff 20 cf ff 20 d2 ff c9  e7
c4c0 : 0d d0 f6 20 cc ff a9 91  8d
c4c8 : 4c d2 ff a0 00 b1 fb 24  57
c4d0 : aa 30 02 50 0c a2 1f dd  2b
c4d8 : 3c c1 f0 2f ca e0 15 d0  c0
c4e0 : f6 a2 04 dd 49 c1 f0 21  8d
c4e8 : dd 4d c1 f0 1e ca d0 f3  5e
c4f0 : a2 38 dd 11 c1 f0 14 ca  d1
c4f8 : e0 16 d0 f6 b1 fb 3d fb  de
c500 : c0 5d 11 c1 f0 05 ca d0  ef
c508 : f3 a2 00 86 ad 8a f0 0f  2e
c510 : a2 11 b1 fb 3d b5 c0 5d  66
c518 : c6 c0 f0 03 ca d0 f3 bd  59
c520 : ea c0 85 ab bd d8 c0 85  f2
c528 : b6 a6 ad 60 a0 01 b1 fb  7a
```

```
c530 : aa c8 b1 fb a0 10 c4 ab  1f
c538 : d0 07 20 4a c5 a0 03 d0  ec
c540 : 02 a4 b6 8e ae 00 8d af  94
c548 : 00 60 a0 01 b1 fb 10 01  fe
c550 : 88 38 65 fb aa e8 f0 01  85
c558 : 88 98 65 fc 60 a2 00 86  4d
c560 : aa 20 64 c2 20 8c c5 a5  54
c568 : ad c9 16 f0 09 c9 30 f0  1f
c570 : 05 c9 21 d0 11 ea 20 94  ce
c578 : c4 20 51 c3 a2 23 a9 2d  5d
c580 : 20 d2 ff ca d0 fa 20 5d  83
c588 : c4 90 d9 60 a2 2c 20 40  a3
c590 : c3 20 23 c3 20 4c c3 20  58
c598 : 75 c6 20 cb c4 20 4c c3  f8
c5a0 : b1 fb 20 2a c3 20 4c c3  92
c5a8 : c8 c4 b6 d0 f3 a9 03 38  a3
c5b0 : e5 b6 aa f0 09 20 49 c3  f7
c5b8 : 20 4c c3 ca d0 f7 a9 20  fc
c5c0 : 20 d2 ff a0 00 a6 ad d0  eb
c5c8 : 11 a2 03 a9 2a 20 d2 ff  0f
c5d0 : ca d0 f8 24 aa 30 85 4c  a0
c5d8 : 6a c6 24 aa 50 29 a9 08  09
c5e0 : 24 ab f0 23 b1 fb 29 fc  14
c5e8 : 85 ad c8 b1 fb 0a a8 b9  d2
c5f0 : 3c 03 8d ae 00 c8 b9 3c  8d
c5f8 : 03 8d af 00 20 be c6 a4  0a
c600 : b6 20 93 c6 20 cb c4 bd  73
c608 : 5b c1 20 d2 ff bd 93 c1  66
c610 : 20 d2 ff bd cb c1 20 d2  42
c618 : ff a9 20 24 ab f0 03 20  07
c620 : 49 c3 a2 20 a9 04 24 ab  9a
c628 : f0 02 a2 28 8a 20 d2 ff  bc
c630 : 24 ab 50 05 a9 23 20 d2  b8
c638 : ff 20 2c c5 88 f0 16 a9  c7
c640 : 08 24 ab f0 07 a9 4d 20  96
c648 : d2 ff a0 01 b9 ad 00 20  ab
c650 : 2a c3 88 d0 f7 a0 03 b9  9c
c658 : ac c0 24 ab f0 09 b9 af  80
c660 : c0 be b2 c0 20 42 c3 88  78
c668 : d0 ed a5 b6 20 67 c3 38  2c
c670 : e9 01 d0 f8 60 a4 d3 a9  fb
c678 : 20 91 d1 c8 c0 28 90 f9  72
c680 : 60 e4 ab d0 04 05 ad 85  81
c688 : ad 60 b9 ad 00 91 fb d1  a9
c690 : fb d0 04 88 10 f4 60 68  00
c698 : 68 60 d0 1c 8a 05 ab 85  72
c6a0 : ab a9 04 85 b5 20 cf ff  6d
c6a8 : c9 20 f0 0d c9 24 f0 09  f3
c6b0 : c9 28 f0 05 c9 2c f0 01  2e
c6b8 : 60 c6 b5 d0 e8 60 e0 18  48
c6c0 : 30 0e ad ae 00 38 e9 02  a6
c6c8 : 38 e5 fb 8d ae 00 a0 40  91
c6d0 : 60 20 7e c2 85 fd a5 fc  11
c6d8 : 85 fe 20 51 c3 20 e4 c6  6d
c6e0 : 30 fb 10 f6 a9 00 85 d3  49
c6e8 : 20 4c c3 20 23 c3 20 4c  8d
c6f0 : c3 20 cf ff a9 01 85 d3  17
c6f8 : a2 80 d0 05 a2 80 8e b1  7b
c700 : 02 86 aa 20 7e c2 a9 25  e3
c708 : 85 c8 2c b1 02 10 08 a2  39
c710 : 0a 20 cf ff ca d0 fa a9  90
c718 : 00 8d b1 02 20 a1 c6 c9  49
c720 : 46 d0 16 46 aa 68 68 a2  f2
c728 : 02 b5 fa 48 b5 fc 95 fa  5c
c730 : 68 95 fc ca d0 f3 4c 64  a2
c738 : c5 c9 2e d0 11 20 9a c2  89
c740 : a0 00 91 fb d1 fb d0 04  0c
c748 : 20 67 c3 c8 88 60 a2 fd  38
c750 : c9 4d d0 19 20 f0 c2 a0  3a
c758 : 00 c9 3f b0 ef 0a a8 a5  60
c760 : fb 99 3c 03 a5 fc c8 99  30
c768 : 3c 03 20 a1 c6 95 a9 e0  e4
c770 : fd d0 04 a9 07 85 b7 e8  59
c778 : d0 f0 a2 38 a5 a6 dd 5b  2e
c780 : c1 f0 05 ca d0 f6 ca 60  05
c788 : a5 a7 dd 93 c1 d0 f4 a5  ac
c790 : a8 dd cb c1 d0 ed bd 11  e7
c798 : c1 85 ad 20 a1 c6 a0 00  5e
c7a0 : e0 20 10 09 c9 20 d0 08  a7
c7a8 : bd 4d c1 85 ad 4c 31 c8  c0
c7b0 : a0 08 c9 4d f0 20 a0 40  83
c7b8 : c9 23 f0 1a 20 9d c2 8d  a7
c7c0 : ae 00 8d af 00 20 a1 c6  dd
c7c8 : a0 20 c9 30 90 1b c9 47  88
c7d0 : b0 17 a0 80 c6 d3 20 a1  13
c7d8 : c6 20 9d c2 8d ae 00 20  fd
c7e0 : a1 c6 c0 08 f0 03 20 be  3b
c7e8 : c6 84 ab a2 01 c9 58 20  2f
c7f0 : 9a c6 a2 04 c9 29 20 9a  b2
c7f8 : c6 a2 02 c9 59 20 9a c6  58
c800 : a5 ad 29 0d f0 0a a2 40  d2
c808 : a9 08 20 81 c6 a9 18 2c  60
c810 : a9 1c a2 82 20 81 c6 a0  2b
c818 : 08 a5 ad c9 20 f0 09 be  c3
c820 : 03 c2 b9 0b c2 20 81 c6  15
c828 : 88 d0 f4 a5 ab 10 01 c8  db
c830 : c8 20 8a c6 c6 b7 a5 b7  b4
c838 : 85 d3 4c 97 c5 20 8d c2  c6
c840 : 8d af 02 60 20 8d c2 8d  c6
c848 : b0 02 60 4c d1 c2 a0 02  55
c850 : 84 bc 88 84 b9 84 bb 88  a5
c858 : 84 b7 20 ca c2 c9 22 d0  be
c860 : ea 20 ca c2 91 bb c8 e6  4d
c868 : b7 c9 22 d0 f4 c6 b7 ad  66
c870 : b0 02 85 ba a5 ac c9 53  67
c878 : f0 13 20 c2 c2 f0 09 a2  6f
c880 : c3 20 80 c2 a9 00 85 b9  f0
c888 : a9 00 6c 30 03 a2 c1 20  df
c890 : 80 c2 a2 ae 20 80 c2 6c  da
c898 : 32 03 20 7e c2 20 ca c2  02
c8a0 : 49 02 4a 4a 08 20 80 c2  cf
c8a8 : 20 51 c3 28 b0 0c a5 fd  65
c8b0 : 65 fb aa a5 fe 65 fc 38  f2
c8b8 : b0 09 a5 fb e5 fd aa a5  1a
c8c0 : fc e5 fe a8 8a 84 fc 85  4f
c8c8 : fb 84 62 85 63 08 a9 00  6c
c8d0 : 85 d3 20 75 c6 a5 fc d0  25
c8d8 : 0f 20 49 c3 a5 fb 20 2a  d1
c8e0 : c3 a5 fb 20 d0 c3 f0 03  6e
c8e8 : 20 23 c3 20 4c c3 a2 90  1d
c8f0 : a5 01 8d b1 02 a9 37 85  05
c8f8 : 01 28 20 49 bc 20 dd bd  fe
c900 : ae b1 02 86 01 4c 56 c3  2c
c908 : 20 8d c2 aa a4 d3 b1 d1  48
c910 : 49 20 f0 a3 8a a8 20 9a  bd
c918 : c2 38 b0 a9 20 b8 c2 a0  6c
c920 : 08 48 20 ca c2 c9 31 68  be
c928 : 2a 88 d0 f5 f0 eb 20 b8  e9
c930 : c2 a2 00 8a 86 fb 85 fc  ed
c938 : a8 20 cf ff c9 3a b0 84  1e
c940 : e9 2f b0 04 38 4c c4 c8  f8
c948 : 85 fd 06 fb 26 fc a5 fc  a8
c950 : 85 fe a5 fb 0a 26 fe 0a  1f
c958 : 26 fe 18 65 fb 08 18 65  db
c960 : fd aa a5 fe 65 fc 28 69  ad
c968 : 00 4c 34 c9 20 7a c2 a9  09
c970 : 37 85 01 a2 04 bd 87 c0  cc
c978 : 95 aa ca 10 f8 20 51 c3  74
c980 : a6 aa a5 ab 20 cd bd e6  8f
c988 : aa d0 02 e6 ab a9 44 20  51
c990 : d2 ff a9 c1 20 d2 ff a0  de
c998 : 00 b1 fb 84 62 85 63 20  20
c9a0 : d1 bd 20 63 c4 a2 03 b0  93
c9a8 : 0a a9 2c a6 d3 e0 49 90  f1
c9b0 : e3 a2 09 86 c6 bd 7d c0  c9
c9b8 : 9d 76 02 ca d0 f7 4c 6e  45
c9c0 : c3 20 7a c2 20 8d c2 a2  49
c9c8 : 00 81 fb 48 20 63 c4 68  92
c9d0 : 90 f7 60 20 5a c2 a5 a6  18
c9d8 : d0 02 c6 a7 c6 a6 20 30  d2
c9e0 : ca 86 b5 a0 02 90 04 a2  69
c9e8 : 02 a0 00 18 a5 a6 65 ae  c0
c9f0 : 85 aa a5 a7 65 af 85 ab  6a
c9f8 : a1 a4 81 a8 41 a8 05 b5  3a
ca00 : 85 b5 a5 a4 c5 a6 a5 a5  d1
ca08 : e5 a7 b0 1d 18 b5 a4 79  45
ca10 : 6b c0 95 a4 b5 a5 79 6c  1c
ca18 : c0 95 a5 8a 18 69 04 aa  90
ca20 : c9 07 90 e8 e9 08 aa b0  99
ca28 : cf a5 b5 f0 0f 4c d1 c2  75
ca30 : 38 a2 fe b5 aa f5 a6 95  50
ca38 : b0 e8 d0 f7 60 20 62 ca  b5
ca40 : 4c d6 c9 4c 62 ca c5 a7  d6
ca48 : d0 02 e4 a6 b0 13 c5 a5  2d
ca50 : d0 02 e4 a4 90 0b 85 b4  d0
ca58 : 8a 18 65 ae aa a5 b4 65  93
ca60 : af 60 20 5a c2 20 7a c2  2f
ca68 : 20 30 ca 20 cb c4 c8 a9  b0
ca70 : 10 24 ab f0 26 a6 fb a5  6e
ca78 : fc 20 46 ca 86 aa b1 fb  ec
ca80 : 85 b5 20 4a c5 a0 01 20  d7
ca88 : 46 ca ca 8a 18 e5 aa 91  b6
ca90 : fb 45 b5 10 19 20 51 c3  fd
ca98 : 20 23 c3 24 ab 10 0f b1  9a
caa0 : fb aa c8 b1 fb 20 46 ca  c8
caa8 : 91 fb 8a 88 91 fb 20 6a  39
cab0 : c6 20 66 c4 90 b5 60 20  31
cab8 : 64 c2 a2 27 20 40 c3 20  5e
cac0 : 23 c3 a0 08 a2 00 20 4c  31
cac8 : c3 a1 fb 20 39 c4 d0 f9  50
cad0 : a2 00 20 5d c4 f0 03 4c  9f
cad8 : ba ca 60 20 7e c2 a0 03  9a
cae0 : 20 cf ff 88 d0 fa 20 ca  f4
cae8 : c2 c9 2e f0 02 91 fb c8  67
caf0 : c0 20 90 f2 60 20 7a c2  b9
caf8 : a2 00 a1 fb c1 fd d0 0b  e7
cb00 : 20 67 c3 e6 fd d0 f3 e6  a5
cb08 : fe d0 ef 20 4c c3 4c c3  c9
cb10 : c3 a9 ff a2 04 95 fa ca  6a
cb18 : d0 fb 20 ca c2 a2 05 dd  58
cb20 : 6e c0 f0 45 ca d0 f8 86  f7
cb28 : a9 20 b4 cb e8 20 cf ff  57
cb30 : c9 20 f0 f3 c9 2c d0 03  0b
cb38 : 20 7a c2 20 51 c3 a4 a9  63
cb40 : b1 fb 20 d6 cb d0 18 88  86
cb48 : 10 f6 20 23 c3 20 4c c3  36
cb50 : a4 d3 c0 24 90 09 20 94  8d
cb58 : c4 20 72 c4 20 51 c3 20  3d
cb60 : 63 c4 90 da a0 27 4c 96  46
cb68 : c4 bd 73 c0 85 a8 bd 78  85
cb70 : c0 85 a9 aa f0 06 20 b4  dc
cb78 : cb ca d0 fa 20 7a c2 20  5d
cb80 : cb c4 20 2c c5 a5 a8 24  af
cb88 : ab d0 09 a8 d0 21 a5 ad  fb
cb90 : d0 1d f0 0d a4 a9 b9 ad  a6
cb98 : 00 20 d6 cb d0 11 88 d0  31
cba0 : f5 84 aa 20 8c c5 20 6f  dc
cba8 : c4 20 66 c4 90 d1 60 20  08
cbb0 : 6a c6 f0 f5 20 c0 cb 9d  eb
cbb8 : cc 03 bd 3c 03 9d 6c 03  d2
cbc0 : 20 ca c2 a0 0f c9 2a d0  94
cbc8 : 02 a0 00 20 af c2 9d 3c  1e
cbd0 : 03 98 9d 9c 03 60 85 b4  cd
cbd8 : 4a 4a 4a 4a 59 6c 03 39  9b
```

Listing 1. »SMON-komplett«-Hauptprogramm. Bitte mit dem MSE eingeben.

```
cbe0 : cc 03 29 0f d0 0a a5 b4  b7
cbe8 : 59 3c 03 39 9c 03 29 0f  ec
cbf0 : 60 68 68 20 cf ff c9 57  75
cbf8 : d0 03 4c 56 cd c9 42 d0  fe
cc00 : 03 4c d0 cd c9 51 d0 03  87
cc08 : 4c 4f cd c9 53 f0 03 4c  0a
cc10 : d1 c2 20 8d c2 48 20 8d  06
cc18 : c2 48 20 49 c2 a0 00 b1  c4
cc20 : fb 8d bc 02 98 91 fb a9  ab
cc28 : 36 8d 16 03 a9 cc 8d 17  70
cc30 : 03 a2 fc 4c ec c3 a2 03  ca
cc38 : 68 9d aa 02 ca 10 f9 68  40
cc40 : 68 ba 8e ae 02 ad a8 02  b3
cc48 : 85 fc ad a9 02 85 fb ad  83
cc50 : bc 02 a0 00 91 fb a9 14  fd
cc58 : 8d 16 03 a9 c2 8d 17 03  e1
cc60 : a9 52 4c ff c2 20 51 c3  3f
cc68 : ad 11 d0 09 10 8d 11 d0  46
cc70 : 60 8d ab 02 08 68 29 ef  0a
cc78 : 8d aa 02 8e ac 02 8c ad  15
cc80 : 02 68 18 69 01 8d a9 02  11
cc88 : 68 69 00 8d a8 02 a9 80  99
cc90 : 8d bc 02 d0 10 20 e5 cd  4b
cc98 : 20 dd fd d8 a2 05 68 9d  70
cca0 : a8 02 ca 10 f9 ad 14 03  61
cca8 : 8d bb 02 ad 15 03 8d ba  5e
ccb0 : 02 ba 8e ae 02 58 ad aa  78
ccb8 : 02 29 10 f0 08 20 65 cc  22
ccc0 : a9 52 4c ff c2 2c bc 02  2a
ccc8 : 50 1f 38 ad a9 02 ed bd  49
ccd0 : 02 8d b1 02 ad a8 02 ed  49
ccd8 : be 02 0d b1 02 d0 67 ad  b0
cce0 : bf 02 d0 5f a9 80 8d bc  0f
cce8 : 02 30 12 4e bc 02 90 cd  0a
ccf0 : ae ae 02 9a a9 cc 48 a9  3f
ccf8 : 70 48 4c ba cd 20 65 cc  04
cd00 : a9 a8 85 fb a9 02 85 fc  99
cd08 : 20 4c c3 a0 00 b1 fb 20  11
cd10 : 2a c3 c8 c0 07 f0 09 c0  04
cd18 : 01 f0 f2 20 4c c3 d0 ed  54
cd20 : ad a9 02 ae a8 02 85 fb  a1
cd28 : 86 fc 20 49 c3 20 cb c4  54
cd30 : 20 c7 c5 20 e4 ff f0 fb  b3
cd38 : c9 4a d0 0a a9 01 8d bc  ee
cd40 : 02 d0 2f ce bf 02 a5 91  16
cd48 : c9 7f d0 26 4c bd cc 20  f0
cd50 : f2 cd a9 40 d0 0a 20 f2  5f
cd58 : cd 08 68 8d aa 02 a9 80  57
cd60 : 8d bc 02 ba 8e ae 02 20  ca
cd68 : 49 c2 20 65 cc ad bc 02  f8
cd70 : f0 37 a2 00 ad 11 d0 a8  9c
cd78 : 29 10 f0 10 98 29 ef 8d  95
cd80 : 11 d0 ea ea a0 0c ca d0  48
cd88 : fd 88 d0 fa 78 a9 48 8d  6e
cd90 : 04 dc 8e 05 dc ad 0e dc  74
cd98 : 29 80 09 11 8d 0e dc a9  76
cda0 : 95 a2 cc 8d bb 02 8e ba  e7
cda8 : 02 ae ae 02 9a 78 ad bb  89
cdb0 : 02 ae ba 02 8d 14 03 8e  9b
cdb8 : 15 03 ad a8 02 48 ad a9  3c
cdc0 : 02 48 ad aa 02 48 ad ab  17
cdc8 : 02 ae ac 02 ac ad 02 40  4d
cdd0 : 20 8d c2 8d be 02 20 8d  b1
cdd8 : c2 8d bd 02 20 8d c2 8d  a5
cde0 : bf 02 4c d6 c2 ad b8 02  0f
cde8 : ae b9 02 8d 14 03 8e 15  63
cdf0 : 03 60 ad 14 03 ae 15 03  11
cdf8 : 8d b8 02 8e b9 02 a9 95  b1
ce00 : 8d 16 03 a9 cc 8d 17 03  2a
ce08 : 60 a9 07 8d 20 d0 a9 36  4c
ce10 : 85 01 a2 00 bd e4 cf 9d  3c
ce18 : d0 02 e8 e0 0d 90 f5 a2  b2
ce20 : 2a 20 40 c3 20 cf ff c9  f7
ce28 : 2a f0 f9 a2 06 dd d2 cf  d7
ce30 : d0 11 8e c1 02 8a 0a aa  57
ce38 : e8 bd d8 cf 48 ca bd d8  b2
ce40 : cf 48 60 ca 10 e7 4c 1f  54
ce48 : ce a9 00 85 fb a9 bf 85  b3
ce50 : fc 85 fe a5 fb 69 04 85  a9
ce58 : fd 20 fc c3 20 e1 ff f0  10
ce60 : 0f ad 8d 02 f0 f6 a9 00  57
ce68 : 85 c6 a5 fc c9 c0 90 e3  06
ce70 : 4c 1f ce 20 7e c2 a0 20  c4
ce78 : a2 00 20 ca c2 20 9a c2  99
ce80 : 81 fb 20 39 c4 d0 f3 20  11
ce88 : 51 c3 4c 24 ce 20 55 cf  35
ce90 : ad c1 02 c9 02 d0 03 4c  23
ce98 : eb ce a2 00 bd 00 bf 8d  89
cea0 : c3 02 e8 bd 00 bf 8d c4  14
cea8 : 02 8a 4c cb ce 20 c2 c2  fa
ceb0 : d0 03 4c 8d ce 20 8d c2  70
ceb8 : 8d c3 02 20 8d c2 8d c4  5a
cec0 : 02 20 55 cf ad c1 02 c9  a6
cec8 : 02 f0 20 20 0d cf a2 0d  42
ced0 : 20 c6 ff a0 00 20 cf ff  a7
ced8 : ea ea ea ea 99 00 bf c8  79
cee0 : d0 f3 20 cc ff 20 bc cf  df
cee8 : 4c 49 ce 20 40 cf a2 0d  b8
cef0 : 20 c9 ff a0 00 b9 00 bf  56
cef8 : 20 d2 ff a6 90 d0 03 c8  83
cf00 : d0 f3 20 cc ff a9 32 20  c2
cf08 : 0d cf 4c b6 cf 8d d1 02  9b
cf10 : ad c3 02 20 79 cf 8e d8  25
cf18 : 02 8d d9 02 ad c4 02 20  e1
cf20 : 79 cf 8e db 02 8d dc 02  a4
cf28 : a2 0f 20 c9 ff a2 00 bd  24
cf30 : d0 02 20 d2 ff e8 e0 0d  49
cf38 : 90 f5 20 cc ff 4c 8c cf  99
cf40 : a2 0f 20 c9 ff a2 00 bd  3c
cf48 : f2 cf 20 d2 ff e8 e0 08  5f
cf50 : 90 f5 4c cc ff a9 0f a8  62
cf58 : a2 08 20 ba ff a9 00 20  eb
cf60 : bd ff 20 c0 ff a9 0d a8  10
cf68 : a2 08 20 ba ff a9 01 a2  04
cf70 : f1 a0 cf 20 bd ff 4c c0  38
cf78 : ff a2 30 38 e9 0a 90 03  13
cf80 : e8 b0 f9 69 3a 60 20 8c  ac
cf88 : cf 4c b6 cf a9 00 85 90  f7
cf90 : 20 51 c3 a9 08 20 b4 ff  d3
cf98 : a9 6f 20 96 ff 20 a5 ff  6b
cfa0 : c9 30 d0 06 4c ab ff 20  d9
cfa8 : a5 ff 20 d2 ff c9 0d d0  d3
cfb0 : f6 20 ab ff 68 68 20 bc  65
cfb8 : cf 4c 1f ce a9 0d 20 c3  5a
cfc0 : ff a9 0f 4c c3 ff a9 06  d0
cfc8 : 8d 20 d0 a9 37 85 01 4c  0b
cfd0 : d6 c2 3a 52 57 4d 58 40  a2
cfd8 : 72 ce ac ce ac ce 48 ce  b6
cfe0 : c5 cf 85 cf 55 31 3a 31  12
cfe8 : 33 20 30 20 31 38 20 30  f1
cff0 : 30 23 42 2d 50 20 31 33  19
cff8 : 20 30 00 ff 67 ff 7f fe  a3
```

Listing 1. (Schluß)

```
Name : ndisass                ce09 cf3e
---------------------------------------
ce09 : 2b 4b 6b 8b 9b ab bb cb  c4
ce11 : eb 89 93 9f 0b 9c 9e 4e  46
ce19 : 53 52 53 52 53 4c 44 49  f0
ce21 : 43 4f 4c 4c 52 52 41 41  e8
ce29 : 43 53 52 50 4f 41 45 41  4b
ce31 : 58 58 50 43 41 25 26 20  48
ce39 : 21 82 80 81 22 21 82 81  24
ce41 : 03 13 07 17 1b 0f 1f 97  48
ce49 : d7 bf df 02 02 02 02 03  76
ce51 : 03 03 02 02 03 03 a2 02  6e
ce59 : d0 28 a6 ad d0 2b a2 01  90
ce61 : b1 fb c9 9c f0 38 c9 80  0f
ce69 : f0 ec c9 89 f0 e8 29 0f  8c
ce71 : c9 02 f0 16 c9 0a f0 0a  ff
ce79 : e8 c9 04 f0 05 e8 c9 0c  3c
ce81 : d0 1c 86 b6 a2 01 8e c5  d0
ce89 : 02 60 b1 fb 29 90 49 80  e4
ce91 : d0 04 a2 02 d0 ec 86 b6  48
ce99 : a2 0a 8e c5 02 60 a0 02  46
cea1 : 84 b6 a0 00 8c c5 02 b1  0b
cea9 : fb a2 0f dd 08 ce f0 d9  e3
ceb1 : ca d0 f8 29 01 f0 d2 b1  8d
ceb9 : fb 4a 4a 4a 4a 4a 18 69  df
cec1 : 02 8d c5 02 a2 0b b1 fb  7d
cec9 : 3d 40 ce dd 40 ce f0 03  da
ced1 : ca d0 f3 bd 35 ce 85 ab  ef
ced9 : bd 4b ce 85 b6 60 a0 00  91
cee1 : a6 ad f0 06 20 4c c3 4c  67
cee9 : da c5 ae c5 02 d0 06 20  09
cef1 : 4c c3 4c c9 c5 a9 2a 20  fe
cef9 : d2 ff bd 17 ce 20 d2 ff  56
cf01 : bd 21 ce 20 d2 ff bd 2b  81
cf09 : ce 4c 16 c6 a9 00 8d 6b  03
cf11 : c0 8d 6c c0 a9 4c 8d 29  50
cf19 : c5 8d be c5 a9 20 8d 30  3f
cf21 : cd a9 5b 8d 2a c5 a9 ce  60
cf29 : 8d 2b c5 a9 df 8d bf c5  e7
cf31 : 8d 31 cd a9 ce 8d c0 c5  e7
cf39 : 8d 32 cd 00 ff 8c cf a2  3c
```

Listing 2. Mit dieser Erweiterung lassen sich illegale Opcodes disassemblieren

```
Name : floppymon              cbf1 cdee
---------------------------------------
cbf1 : a9 36 85 01 a2 00 bd b2  bd
cbf9 : cd f0 06 20 d2 ff e8 d0  36
cc01 : f5 20 51 c3 a2 3e 20 40  f0
cc09 : c3 20 cf ff c9 3e f0 f9  16
cc11 : c9 20 f0 f5 a2 05 dd c8  40
cc19 : cd f0 09 ca d0 f8 20 51  f2
cc21 : c3 4c 05 cc 8a 0a aa e8  5b
cc29 : bd cc cd 48 ca bd cc cd  32
cc31 : 48 60 20 c2 c2 d0 0a a9  38
cc39 : 00 8d c0 cd 8d c1 cd f0  ea
cc41 : 1a 20 8d c2 8d c1 cd 20  85
cc49 : c2 c2 d0 07 a9 00 8d c0  d4
cc51 : cd f0 08 20 8d c2 29 f8  22
cc59 : 8d c0 cd 20 77 cd a2 0f  4c
cc61 : 20 c9 ff a2 00 bd bd cd  3a
cc69 : 20 d2 ff e8 e0 06 90 f5  7c
```

Listing 3. Komfortabler Disketten-Monitor für den SMON. Bitte mit dem MSE eingeben.

```
cc71 : 20 cc ff a2 0f 20 c6 ff  59
cc79 : a0 00 20 cf ff 99 00 bf  68
cc81 : c8 d0 f7 20 cc ff 4c bc  2b
cc89 : cf a9 bf 85 fc a9 00 85  f5
cc91 : fb 60 20 33 cc 20 8a cc  bc
cc99 : a2 3a 20 40 c3 ad c1 cd  b5
cca1 : 20 2a c3 ad c0 cd 20 2a  cc
cca9 : c3 a0 20 a2 00 20 4c c3  d2
ccb1 : 20 4c c3 a1 fb 20 2a c3  0d
ccb9 : a1 fb 20 39 c4 d0 f1 a9  75
ccc1 : 08 18 6d c0 cd 8d c0 cd  31
ccc9 : 08 c9 f8 d0 06 20 5c cc  7a
ccd1 : 20 8a cc 28 90 09 ee c1  ff
ccd9 : cd 20 5c cc 20 8a cc 20  31
cce1 : 87 cd 20 e1 ff d0 b1 4c  79
cce9 : 02 cc 20 7e c2 a5 fb 8d  8e
ccf1 : c6 cd a5 fc 8d c7 cd 20  35
ccf9 : 19 cd a0 20 a2 00 20 ca  65
cd01 : c2 20 ca c2 20 9a c2 20  00
cd09 : d2 ff 20 39 c4 d0 f2 20  e9
cd11 : cc ff 20 bc cf 4c 07 cc  92
cd19 : 20 77 cd a2 0f 20 c9 ff  d6
cd21 : a2 00 bd c3 cd 20 d2 ff  d4
cd29 : e8 e0 06 90 f5 60 20 7e  f5
cd31 : c2 a2 fd 20 80 c2 a5 fd  78
cd39 : 8d c6 cd a5 fe 8d c7 cd  68
cd41 : a9 20 8d c8 cd 20 19 cd  55
cd49 : a0 00 b1 fb 20 d2 ff c8  ff
cd51 : c0 20 90 f6 18 a9 20 6d  4e
cd59 : c6 cd b0 0c 8d c6 cd a9  4d
cd61 : 20 65 fb 85 fb 4c 46 cd  ba
cd69 : 20 cc ff 20 bc cf a9 08  f4
cd71 : 8d c8 cd 4c 02 cc a9 0f  ab
cd79 : a8 a2 08 20 ba ff a9 00  cb
cd81 : 20 bd ff 4c c0 ff 20 e4  60
cd89 : ff f0 fb 60 20 c2 c2 d0  d0
cd91 : 03 4c 86 cf a9 08 20 b1  15
cd99 : ff a9 6f 20 93 ff 20 cf  a6
cda1 : ff 20 a8 ff c9 0d d0 f6  11
cda9 : 20 ae ff 4c 02 cc 4c 09  74
cdb1 : ce 0d 3e 46 4c 4f 50 2d  39
cdb9 : 4d 4f 4e 00 4d 2d 52 00  c9
cdc1 : 00 ff 4d 2d 57 00 00 08  3f
cdc9 : 3a 4d 56 40 58 ea cc 92  7d
cdd1 : cc 2e cd 8c cd ae cd a9  96
cdd9 : 00 8d 22 c0 a9 46 8d d7  f3
cde1 : cf a9 cb 8d e3 cf a9 f0  6f
cde9 : 8d e2 cf 00 00 8e 15 03  aa
```

Listing 3. Komfortabler Disketten-Monitor für den SMON. Bitte mit dem MSE eingeben.

```
Name : illegal-code        4000 40f7
-------------------------------------
4000 : 87 87 c7 c7 e7 e7 a7 a7  e1
4008 : 27 27 67 67 07 07 47 47  de
4010 : d7 d7 f7 ff 37 aa 17 60  b6
4018 : 57 20 97 13 b7 20 8f 0f  a0
4020 : cf cf 01 8f ef 20 0c af  99
4028 : 19 20 2f 24 30 6f 60 60  62
4030 : 0f 0c 04 4f 20 2a df 05  0d
4038 : 06 ff 0f 04 3f 60 60 7f  fa
4040 : 03 0d 1f 20 23 5f 32 30  ec
4048 : db 60 20 fb 05 11 bf 01  b5
4050 : 04 3b 03 0f 7b 12 33 1b  e0
4058 : 60 01 5b 12 03 83 12 c3  6e
4060 : 32 e3 0c a3 18 23 01 63  61
4068 : 01 03 31 43 31 d3 19 f3  9e
4070 : 60 b3 0c 33 01 73 01 13  e9
4078 : 01 53 32 53 32 ea 6d 54  f3
4080 : 6b 03 81 09 a6 07 b9 02  7c
4088 : 43 27 9d 06 ae 07 a1 1a  65
4090 : 5b 3f 85 1e b6 0f a9 12  5f
4098 : 53 37 8d 16 be 2f 91 2a  ad
40a0 : 6b 0f b5 2e 86 67 99 22  14
40a8 : 63 07 bd 26 8e 67 81 3a  62
40b0 : 7b 1f a5 3e 9b 43 b2 21  cd
40b8 : 57 bd b6 b0 4a b5 20 43  0b
40c0 : ef 36 ce 95 59 ee 49 c4  ec
40c8 : 3f c1 ee 4d e7 3a f0 54  0a
40d0 : 72 d1 00 f7 1e f7 f0 f0  71
40d8 : 5a 7b aa 60 fc f7 c9 60  1e
40e0 : 42 95 59 05 cb f2 ea e4  ae
40e8 : 92 a4 e1 94 a1 c2 03 fb  0b
40f0 : 00 54 20 00 00 01 08 18  7b
```

Listing 4. Mit dem Befehl D 4000 erscheinen alle illegalen Opcodes disassembliert auf dem Bildschirm

```
100 REM ********************************     <238>
110 REM *                              *     <159>
120 REM *       SMON - ERWEITERUNG     *     <222>
130 REM *                              *     <179>
140 REM *  VON   MARK RICHTERS         *     <005>
150 REM *        ALLERSTR.4            *     <068>
160 REM *        2806 OYTEN            *     <037>
170 REM *        TEL.: 04207/1870      *     <078>
180 REM *                              *     <229>
190 REM ********************************     <072>
200 :                                        <176>
210 PRINT"BITTE DIE STARTADRESSE"            <248>
220 PRINT"IHRER SMON-VERSION EINGEBEN!"      <189>
230 INPUT H :AL=H:H=H/4096                   <234>
240 IF H<>INT(H) THEN 210                    <052>
250 POKE AL+38,0                             <239>
260 :                                        <238>
270 DIM W(75)                                <045>
280 FOR I=0 TO 9                             <096>
290 :  W(48+I)=I                             <224>
300 :  W(65+I)=I+10                          <069>
310 NEXT I                                   <140>
320 :                                        <042>
330 FOR I=1 TO 4                             <048>
340 :  READ K,Z : K=K+H*4096                 <123>
350 :  FOR J=K TO K+Z-1                      <235>
360 :    READ A$                             <038>
370 :    A=W(ASC(A$))                        <253>
380 :    B=W(ASC(RIGHT$(A$,1)))              <129>
390 :    S=S+A+B                             <023>
400 :    IF ASC(A$)=42 THEN A=H              <010>
410 :    P=A*16+B : POKE J,P                 <118>
420 :  NEXT J                                <070>
430 NEXT I                                   <004>
440 :                                        <162>
450 IF S=7031 THEN PRINT"OK!":GOTO 470       <200>
460 PRINT"FEHLER IN DATAS !"                 <071>
470 STOP                                     <026>
480 :                                        <202>
490 DATA 97,7                                <136>
500 DATA 27,3A,3B,2C,28,29,21                <025>
510 :                                        <232>
520 DATA 739,28                              <137>
530 DATA A2,06,DD,61,*0,F0,08,CA             <159>
540 DATA 10,F8,A9,2E,20,D2,FF,20             <187>
550 DATA CA,*2,C9,2E,F0,F9,4C,85             <184>
560 DATA*F,4C,B2,*F                          <109>
570 :                                        <038>
580 DATA 781,2                               <220>
590 DATA F0,ED                               <141>
600 :                                        <068>
610 DATA 3593,503                            <202>
620 DATA A9,80,2C,A9,00,85,AB,20             <085>
630 DATA 64,*2,24,AB,10,03,A2,29             <117>
640 DATA 2C,A2,28,20,40,*3,20,23             <239>
650 DATA*3,A0,06,A2,00,A1,FB,0A              <071>
660 DATA 48,B0,03,A9,2E,2C,A9,2A             <144>
670 DATA 91,D1,AD,86,02,91,F3,68             <079>
680 DATA C8,E8,E0,08,D0,E9,20,67             <139>
690 DATA*3,24,AB,30,04,C0,1E,90              <233>
700 DATA DA,20,5D,*4,90,C4,60,A0             <133>
710 DATA 08,2C,A0,18,20,7E,*2,20             <100>
720 DATA B8,*2,A2,08,A9,00,85,AA             <236>
730 DATA 20,CA,*2,C9,2E,F0,07,C9             <052>
740 DATA 2A,F0,04,4C,D1,*2,18,26             <202>
750 DATA AA,88,CA,D0,EB,A5,AA,81             <014>
760 DATA FB,C1,FB,D0,EE,20,67,*3             <071>
770 DATA C0,00,D0,D6,60,A9,80,2C             <218>
780 DATA A9,00,85,AB,20,64,*2,20             <243>
790 DATA 51,*3,24,AB,10,0B,A9,21             <138>
```

Listing 5. Erweiterung zum SMON. Bitte mit dem Checksummer eingeben.

```
800 DATA 20,D2,FF,20,23,*3,A0,08      <033>
810 DATA 2C,A0,00,A2,00,A1,FB,20      <242>
820 DATA 4F,*4,D0,F9,20,5D,*4,90      <059>
830 DATA DE,60,20,7E,*2,A2,00,A0      <137>
840 DATA 08,B1,D1,81,FB,C1,FB,D0      <193>
850 DATA AA,20,56,*4,90,F3,60,20      <193>
860 DATA 8D,*2,29,F0,85,FF,20,3C      <190>
870 DATA*F,20,D6,*9,20,3C,*F,A9       <001>
880 DATA 14,85,FB,A9,02,05,FF,85      <195>
890 DATA FC,A9,D1,85,FD,A9,0F,05      <008>
900 DATA FF,85,FE,20,68,*A,A0,00      <182>
910 DATA A2,0D,BD,F2,*F,05,FF,85      <072>
920 DATA FC,CA,BD,F2,*F,85,FB,B1      <238>
930 DATA FB,29,0F,05,FF,91,FB,CA      <150>
940 DATA 10,E8,A9,2B,85,FB,A9,00      <089>
950 DATA 05,FF,85,FC,A0,35,B1,FB      <082>
960 DATA 29,0F,05,FF,91,FB,88,88      <137>
970 DATA 10,F4,A9,DE,85,FB,A9,0F      <098>
980 DATA 05,FF,85,FC,A0,13,B1,FB      <088>
990 DATA 29,0F,05,FF,91,FB,88,88      <167>
1000 DATA 10,F4,60,A5,FF,85,A9,20     <096>
1010 DATA 43,*F,68,68,29,F0,85,A5     <117>
1020 DATA 18,69,10,85,A7,A9,00,85     <054>
1030 DATA A4,85,A6,85,A8,60,20,7A     <029>
1040 DATA*2,A9,00,4C,C7,*9,20,7E      <250>
1050 DATA*2,A0,00,A9,D0,84,FD,85      <002>
1060 DATA FE,78,A9,03,85,01,A2,10     <164>
1070 DATA B1,FD,91,FB,C8,D0,F9,E6     <125>
1080 DATA FC,E6,FE,CA,D0,F2,A9,27     <124>
1090 DATA 85,01,58,60,48,C9,4A,D0     <178>
1100 DATA 10,A0,27,B9,00,02,91,D1     <224>
1110 DATA 88,10,F8,68,C6,D6,4C,D6     <214>
1120 DATA*2,A0,06,D9,D7,*F,D0,0A      <187>
1130 DATA A0,27,B1,D1,99,00,02,88     <003>
1140 DATA 10,F8,88,10,EE,68,4C,FF     <033>
1150 DATA*2,A2,0A,DD,D0,*F,F0,06      <139>
1160 DATA CA,D0,F8,4C,D1,*2,20,C5     <228>
1170 DATA*F,4C,D6,*2,8A,0A,AA,BD      <009>
1180 DATA DD,*F,48,BD,DC,*F,48,60     <111>
1190 DATA 28,29,21,45,59,51,48,5A     <209>
1200 DATA 4E,55,44,4B,4D,52,*E,4F     <203>
1210 DATA*E,B2,*E,56,*F,C7,*E,5E      <099>
1220 DATA*F,0B,*E,08,*E,85,*E,88      <157>
1230 DATA*E,06,00,87,03,2D,0C,5C      <076>
1240 DATA 0C,F5,0C,A2,0D,04,0E        <224>
```

© 64'er

Listing 5. Erweiterung zum SMON (Schluß)

```
110:   C026                 .OPT P4,OO
                       ;
                       ; !SMON-ERWEITERUNG
                       ; ERWEITERT SMON UM FOLGENDE
                       ; BEFEHLE    - H ADR1  (ADR2)
                       ;            - Z ADR1  (ADR2)
                       ;            - N ADR1  (ADR2)
                       ;            - U ADR1  (ADR2)
                       ;            - E ADR1   ADR2
                       ;            - Y BYTE
                       ;            - Q ADR1
                       ;            - J
                       ;
                       ;-------------------------------
                       ;
700:   C026            S          =    $C000      ;BASISADRESSE SMON
                       ;
720:   C026            HCH        =    "("        ;HIDDEN COMMANDS
730:   C026            HCZ        =    ")"
740:   C026            HCN        =    "!"
                       ;
760:   C026            FLAG       =    $AA
770:   C026            TEMP       =    $AB
780:   C026            PCL        =    $FB
790:   C026            PCH        =    $FC
800:   C026            HINIBBLE   =    $FF
                       ;
820:   C026            CMDS       =    S+$02B
830:   C026            GET12ADR   =    S+$264
840:   C026            GET2ADR    =    S+$27A
850:   C026            GETADR1    =    S+$27E
860:   C026            GETBYT     =    S+$28D
870:   C026            SKIPSPAC   =    S+$2B8
880:   C026            GETRET     =    S+$2C2
890:   C026            GETCHRER   =    S+$2CA
900:   C026            ERROR      =    S+$2D1
910:   C026            EXECUTE    =    S+$2D6
920:   C026            CMDSTORE   =    S+$2FF
930:   C026            CHARRET    =    S+$340
940:   C026            RETURN     =    S+$351
950:   C026            HEXOUT     =    S+$323
960:   C026            PCINC      =    S+$367
970:   C026            ASCII4     =    S+$44F
980:   C026            ASCII5     =    S+$456
990:   C026            CONTIN     =    S+$45D
1000:  C026            OCCUPY     =    S+$9C7
                       ;--------------
1011:  C026                       *=   S+38
1012:  C026 00                    .BYTE00          ;ALTEN Z-BEFEHL LOESCHEN
                       ;
1020:  C061                       *=   S+$61
                       ;
1040:  C061 27 3A 3B HCMDTAB     .ASC "':;,"
1050:  C065 28 29 21             .BYTEHCH,HCZ,HCN
                       ;
1070:  C2E3                       *=   S+$2E3
                       ;
1090:  C2E3 A2 06                 LDX  #6          ;ZEILENANFANG
1100:  C2E5 DD 61 C0 F1           CMP  HCMDTAB,X
1110:  C2E8 F0 08                 BEQ  EXEC1
1120:  C2EA CA                    DEX
1130:  C2EB 10 F8                 BPL  F1
1140:  C2ED A9 2E                 LDA  #"."
1150:  C2EF 20 D2 FF              JSR  $FFD2
                       ;
1170:  C2F2 20 CA C2 EXEC1        JSR  GETCHRERR
1180:  C2F5 C9 2E                 CMP  #"."
1190:  C2F7 F0 F9                 BEQ  EXEC1
1200:  C2F9 4C 85 CF              JMP  LINSTORE
1210:  C2FC 4C B2 CF WEITER       JMP  MORECMD
                       ;
1230:  C30D                       *=   S+$30D       ;WEITERE BEFEHLE
                       ;
1250:  C30D F0 ED                 BEQ  WEITER
                       ;
1270:  CE09                       *=   S+$E09       ;NEUE ROUTINEN
                       ;
                       ;--------------
                       ;
1310:  CE09 A9 80    ZCMD         LDA  #$80      ;FLAG SETZEN
1320:  CE0B 2C                    .BYTE$2C
                       ;
1340:  CE0C A9 00    HCMD         LDA  #0
1350:  CE0E 85 AB                 STA  TEMP
1360:  CE10 20 64 C2              JSR  GET12ADR   ;START/END-ADRESSE
                       ;
1380:  CE13 24 AB    L1           BIT  TEMP
1390:  CE15 10 03                 BPL  W8
1400:  CE17 A2 29                 LDX  #HCZ        ;HIDDEN COMMAND
1410:  CE19 2C                    .BYTE$2C      ;AUSGEBEN
1420:  CE1A A2 28    W8           LDX  #HCH
1430:  CE1C 20 40 C3              JSR  CHARRET
1440:  CE1F 20 23 C3              JSR  HEXOUT      ;PC AUSGEBEN
1450:  CE22 A0 06                 LDY  #6          ;SPALTE 6
1470:  CE24 A2 00    L2           LDX  #0
1480:  CE26 A1 FB                 LDA  (PCL,X)
                       ;
1500:  CE28 0A       L3           ASL
1510:  CE29 48                    PHA             ;BYTE MERKEN
1520:  CE2A B0 03                 BCS  BITSET        ;BIT=1, DANN *
1530:  CE2C A9 2E                 LDA  #"."          ;BIT=0, DANN .
1540:  CE2E 2C                    .BYTE$2C      ;AUSGEBEN
1550:  CE2F A9 2A    BITSET       LDA  #"*"
1560:  CE31 91 D1                 STA  ($D1),Y
1570:  CE33 AD 86 02              LDA  $286
1580:  CE36 91 F3                 STA  ($F3),Y
                       ;
1600:  CE38 68                    PLA             ;BYTE ZURUECKHOLEN
1610:  CE39 C8                    INY             ;CURSOR WEITER
1620:  CE3A E8                    INX             ;NAECHSTES BYTE
1630:  CE3B E0 08                 CPX  #8            ;8 BIT
1640:  CE3D D0 E9                 BNE  L3            ;BYTE WEITERSCHIEBEN
                       ;
1660:  CE3F 20 67 C3              JSR  PCINC       ;ZAEHLER ERHOEHEN
1670:  CE42 24 AB                 BIT  TEMP        ;FLAG FUER 1*8 BIT
1680:  CE44 30 04                 BMI  W9          ;ZEILE FERTIG
1690:  CE46 C0 1E                 CPY  #30         ;3 BYTES
1700:  CE48 90 DA                 BCC  L2
1710:  CE4A 20 5D C4 W9           JSR  CONTIN  ;TASTEN-ABFRAGE
1720:  CE4D 90 C4                 BCC  L1
1730:  CE4F 60                    RTS
                       ;
                       ;--------------
                       ;
1770:  CE50 A0 08    ZCMDH        LDY  #8
1780:  CE52 2C                    .BYTE$2C       ;1 BYTE
                       ;
1800:  CE53 A0 18    HCMDH        LDY  #24     ;3 BYTE
1810:  CE55 20 7E C2              JSR  GETADR1
1820:  CE58 20 B8 C2              JSR  SKIPSPACE  ;SPACES UEBERLESEN
1830:  CE5B A2 08    A1           LDX  #8
1840:  CE5D A9 00                 LDA  #0
1850:  CE5F 85 AA                 STA  FLAG
                       ;
1870:  CE61 20 CA C2 A2           JSR  GETCHRERR
1880:  CE64 C9 2E                 CMP  #"."          ;. => BIT=0
1890:  CE66 F0 07                 BEQ  BIT0          ;* => BIT=1
1900:  CE68 C9 2A                 CMP  #"*"
1910:  CE6A F0 04                 BEQ  BIT1
1920:  CE6C 4C D1 C2 ERR1         JMP  ERROR     ;ANDERES ZEICHEN
                       ;
1940:  CE6F 18       BIT0         CLC
1950:  CE70 26 AA    BIT1         ROL  FLAG
1960:  CE72 88                    DEY
1970:  CE73 CA                    DEX            ;BYTE
1980:  CE74 D0 EB                 BNE  A2           ;NOCH NICHT FERTIG
                       ;
2000:  CE76 A5 AA                 LDA  FLAG        ;IN SPEICHER
2010:  CE78 81 FB                 STA  (PCL,X)     ;SCHREIBEN
2020:  CE7A C1 FB                 CMP  (PCL,X)
2030:  CE7C D0 EE                 BNE  ERR1
2040:  CE7E 20 67 C3              JSR  PCINC
                       ;
2060:  CE81 C0 00                 CPY  #0
2070:  CE83 D0 D6                 BNE  A1           ;NOCH NICHT FERTIG
2080:  CE85 60                    RTS
                       ;
                       ;--------------
                       ;
```

```
2120:  CEB6 A9 80     NCMD     LDA  #$80
2130:  CEB8 2C                 .BYTE$2C
2140:  CEB9 A9 00     UCMD     LDA  #0
2150:  CEBB 85 AB              STA  TEMP
2160:  CEBD 20 64 C2           JSR  GET12ADR
                      ;
2180:  CE90 20 51 C3  L5       JSR  RETURN
2190:  CE93 24 AB              BIT  TEMP
2200:  CE95 10 0B              BPL  U
2210:  CE97 A9 21              LDA  #HCN        ;HIDDEN COMMAND
2220:  CE99 20 D2 FF           JSR  $FFD2
2230:  CE9C 20 23 C3           JSR  HEXOUT
2240:  CE9F A0 08              LDY  #8          ;SPALTE 8
2250:  CEA1 2C                 .BYTE$2C
                      ;
2270:  CEA2 A0 00     U        LDY  #0         ;SPALTE 0
2280:  CEA4 A2 00              LDX  #0
                      ;
2300:  CEA6 A1 FB     L4       LDA  (PCL,X)
2310:  CEA8 20 4F C4           JSR  ASCII4      ;ALS BILDSCHIRM-CODE
2320:  CEAB D0 F9              BNE  L4          ;AUSGEBEN
                      ;
2340:  CEAD 20 5D C4           JSR  CONTIN
2350:  CEB0 90 DE              BCC  L5          ;NAECHSTE ZEILE
2360:  CEB2 60                 RTS
                      ;-------------
2380:  CEB3 20 7E C2  NCMDH    JSR  GETADR1
2390:  CEB6 A2 00              LDX  #0
2400:  CEB8 A0 08              LDY  #8          ;SPALTE 8
2410:  CEBA B1 D1     C1       LDA  ($D1),Y
2420:  CEBC 81 FB              STA  (PCL,X)     ;IN SPEICHER
2430:  CEBE C1 FB              CMP  (PCL,X)     ;SCHREIBEN
2440:  CEC0 D0 AA              BNE  ERR1
2450:  CEC2 20 56 C4           JSR  ASCII5      ;PC ERHOEHEN
2460:  CEC5 90 F3              BCC  C1
2470:  CEC7 60                 RTS             ;ZEILE FERTIG
                      ;
                      ;-------------
                      ;
2510:  CEC8 20 8D C2  YCMD     JSR  GETBYT
2520:  CECB 29 F0              AND  #%11110000
2530:  CECD 85 FF              STA  HINIBBLE    ;NEUER 4K-BLOCK
2540:  CECF 20 3C CF           JSR  SETPTR      ;ADRESSEN SETZEN
2550:  CED2 20 D6 79           JSR  $79D6       ;W-BEFEHL
                      ;
2570:  CED5 20 3C CF           JSR  SETPTR
2580:  CED8 A9 14              LDA  #$14        ;BEREICH OHNE
2590:  CEDA 85 FB              STA  $FB         ;TABELLEN
2600:  CEDC A9 02              LDA  #2
2610:  CEDE 05 FF              ORA  HINIBBLE
2620:  CEE0 85 FC              STA  $FC
2630:  CEE2 A9 D1              LDA  #<NEWCMDS
2640:  CEE4 85 FD              STA  $FD
2650:  CEE6 A9 0F              LDA  #(NEWCMDS>8 & $F)
2660:  CEE8 05 FF              ORA  HINIBBLE
2670:  CEEA 85 FE              STA  $FE
2680:  CEEC 20 68 7A           JSR  $7A68       ;V-BEFEHL
                      ;
2700:  CEEF A0 00              LDY  #0          ;LADE-BEFEHLE
2710:  CEF1 A2 0D              LDX  #13         ;ANPASSEN
2720:  CEF3 BD F2 CF  D3       LDA  CHANGE,X
2730:  CEF6 05 FF              ORA  HINIBBLE
2740:  CEF8 85 FC              STA  PCH         ;ADRESSE ALS ZEIGER
2750:  CEFA CA                 DEX
2760:  CEFB BD F2 CF           LDA  CHANGE,X
2770:  CEFE 85 FB              STA  PCL
2780:  CF00 B1 FB              LDA  (PCL),Y
2790:  CF02 29 0F              AND  #%00001111
2800:  CF04 05 FF              ORA  HINIBBLE
2810:  CF06 91 FB              STA  (PCL),Y
2820:  CF08 CA                 DEX
2830:  CF09 10 E8              BPL  D3
                      ;
2850:  CF0B A9 2B              LDA  #<CMDS      ;ZEIGER AUF
2860:  CF0D 85 FB              STA  PCL         ;BEFEHLSADRESSEN
2870:  CF0F A9 00              LDA  #(CMDS>8 & $F)
2880:  CF11 05 FF              ORA  HINIBBLE
2890:  CF13 85 FC              STA  PCH
2910:  CF15 A0 35              LDY  #53
2920:  CF17 B1 FB     D1       LDA  (PCL),Y
2930:  CF19 29 0F              AND  #%00001111
2940:  CF1B 05 FF              ORA  HINIBBLE    ;HIBYTES
2950:  CF1D 91 FB              STA  (PCL),Y     ;ANGLEICHEN
2960:  CF1F 88        W1       DEY
2970:  CF20 88                 DEY
2980:  CF21 10 F4              BPL  D1
                      ;
3000:  CF23 A9 DE              LDA  #<NEWADR    ;ZEIGER AUF NEUE
3010:  CF25 85 FB              STA  PCL         ;BEFEHLSADRESSEN
3020:  CF27 A9 0F              LDA  #(NEWADR>8 & $F)
3030:  CF29 05 FF              ORA  HINIBBLE
3040:  CF2B 85 FC              STA  PCH
                      ;
3060:  CF2D A0 13              LDY  #TABEND-NEWADR-1
3070:  CF2F B1 FB     D2       LDA  (PCL),Y
3080:  CF31 29 0F              AND  #%00001111
3090:  CF33 05 FF              ORA  HINIBBLE    ;HIBYTES
3100:  CF35 91 FB              STA  (PCL),Y     ;ANGLEICHEN
3110:  CF37 88                 DEY
3120:  CF38 88                 DEY
3130:  CF39 10 F4              BPL  D2
3140:  CF3B 60                 RTS
                      ;
                      ;
3170:  CF3C A5 FF     SETPTR   LDA  HINIBBLE
3180:  CF3E 85 A9              STA  $A9         ;NEUER START HI
3190:  CF40 20 43 CF           JSR  GETHI
3200:  CF43 68        GETHI    PLA
3210:  CF44 68                 PLA
3220:  CF45 29 F0              AND  #%11110000
3230:  CF47 85 A5              STA  $A5         ;ALTER START HI
3240:  CF49 18                 CLC
3250:  CF4A 69 10              ADC  #$10
3260:  CF4C 85 A7              STA  $A7         ;ALTES ENDE HI
                      ;
3280:  CF4E A9 00              LDA  #0          ;LO-BYTES LOESCHEN
3290:  CF50 85 A4              STA  $A4
3300:  CF52 85 A6              STA  $A6
3310:  CF54 85 A8              STA  $A8
3320:  CF56 60                 RTS
                      ;
                      ;---------
                      ;
3360:  CF57 20 7A C2  ECMD     JSR  GET2ADR
3370:  CF5A A9 00              LDA  #0
3380:  CF5C 4C C7 C9           JMP  OCCUPY
                      ;
                      ;---------
                      ;
3420:  CF5F 20 7E C2  QCMD     JSR  GETADR1
3430:  CF62 A0 00              LDY  #0
3440:  CF64 A9 D0              LDA  #$D0        ;ZEIGER AUF
3450:  CF66 84 FD              STY  $FD         ;ZEICHENROM
3460:  CF68 85 FE              STA  $FE
                      ;
3480:  CF6A 78                 SEI
3490:  CF6B A9 03              LDA  #%011       ;ROM EINBLENDEN
3500:  CF6D 85 01              STA  1
3510:  CF6F A2 10              LDX  #16         ;4K UEBERTRAGEN
3520:  CF71 B1 FD     E1       LDA  ($FD),Y
3530:  CF73 91 FB              STA  (PCL),Y
3540:  CF75 C8                 INY
3550:  CF76 D0 F9              BNE  E1
3560:  CF78 E6 FC              INC  PCH
3570:  CF7A E6 FE              INC  $FE
3580:  CF7C CA                 DEX
3590:  CF7D D0 F2              BNE  E1
3600:  CF7F A9 27              LDA  #$27        ;NORMALEINSTELLUNG
3610:  CF81 85 01              STA  1
3620:  CF83 58                 CLI
3630:  CF84 60                 RTS
                      ;
                      ;---------
                      ;
3670:  CF85 48        LINSTORE PHA             ;BEFEHL MERKEN
3680:  CF86 C9 4A              CMP  #"J"
3690:  CF88 D0 10              BNE  STORE
                      ;
3710:  CF8A A0 27              LDY  #39
3720:  CF8C B9 00 02  G1       LDA  $0200,Y
3730:  CF8F 91 D1              STA  ($D1),Y     ;ZEILE AUF
3740:  CF91 88                 DEY             ;BILDSCHIRM
3750:  CF92 10 F8              BPL  G1          ;SCHREIBEN
3760:  CF94 68                 PLA
3770:  CF95 C6 D6              DEC  $D6         ;CURSOR 1 HOCH
3780:  CF97 4C D6 C2           JMP  EXECUTE
                      ;
3800:  CF9A A0 06     STORE    LDY  #6
3810:  CF9C D9 D7 CF  G3       CMP  OUTCMDS,Y
3820:  CF9F D0 0A              BNE  W3
                      ;
3840:  CFA1 A0 27     OK1      LDY  #39
3850:  CFA3 B1 D1     G2       LDA  ($D1),Y
3860:  CFA5 99 00 02           STA  $0200,Y     ;ZEILE NUR BEI
3870:  CFA8 88                 DEY             ;H,Z,N,U,K,M,D
3880:  CFA9 10 F8              BPL  G2          ;SPEICHERN
                      ;
3900:  CFAB 88        W3       DEY
3910:  CFAC 10 EE              BPL  G3
3920:  CFAE 68        STEND    PLA             ;BEFEHL ZURUECKHOLEN
3930:  CFAF 4C FF C2           JMP  CMDSTORE
                      ;
                      ;+++++++++
                      ;
3970:  CFB2 A2 0A     MORECMD  LDX  #NEWADR-NEWCMDS-3
3980:  CFB4 DD D0 CF  B1       CMP  NEWCMDS-1,X
3990:  CFB7 F0 06              BEQ  FOUND
4000:  CFB9 CA                 DEX
4010:  CFBA D0 F8              BNE  B1
4020:  CFBC 4C D1 C2           JMP  ERROR
                      ;
4040:  CFBF 20 C5 CF  FOUND    JSR  CMDEXEC2
4050:  CFC2 4C D6 C2           JMP  EXECUTE
                      ;
4070:  CFC5 8A        CMDEXEC2 TXA
4080:  CFC6 0A                 ASL
4090:  CFC7 AA                 TAX
4100:  CFC8 BD DD CF           LDA  NEWADR-1,X
4110:  CFCB 48                 PHA
4120:  CFCC BD DC CF           LDA  NEWADR-2,X
4130:  CFCF 48                 PHA
4140:  CFD0 60                 RTS
                      ;
4160:  CFD1 28 29 21  NEWCMDS  .BYTEHCH,HCZ,HCN
4170:  CFD4 45 59 51           .ASC "EYQ"
4180:  CFD7 48 5A 4E  OUTCMDS  .ASC "HZNUDKM"
                      ;
4200:  CFDE 52 CE     NEWADR   .WORDHCMDH-1
4210:  CFE0 4F CE              .WORDZCMDH-1
4220:  CFE2 B2 CE              .WORDNCMDH-1
4230:  CFE4 56 CF              .WORDECMD-1              ;E
4240:  CFE6 C7 CE              .WORDYCMD-1              ;Y
4250:  CFE8 5E CF              .WORDQCMD-1              ;Q
4260:  CFEA 0B CE              .WORDHCMD-1              ;!
4270:  CFEC 08 CE              .WORDZCMD-1              ;Z
4280:  CFEE 85 CE              .WORDNCMD-1              ;N
4290:  CFF0 88 CE              .WORDUCMD-1              ;U
4300:  CFF2           TABEND   =    *
                      ;
4320:  CFF2 06 00     CHANGE   .WORD$0005+1
4330:  CFF4 87 03              .WORD$0386+1
4340:  CFF6 2D 0C              .WORD$0C2C+1
4350:  CFF8 5C 0C              .WORD$0C5B+1
4360:  CFFA F5 0C              .WORD$0CF4+1
4370:  CFFC A2 0D              .WORD$0DA1+1
4380:  CFFE 04 0E              .WORD$0E03+1
```

Listing 6. Assembler-Listing zur SMON-Erweiterung (Schluß)



Passend zum Assembler Hypra-Ass stellen wir Ihnen einen professionellen Reassembler vor, der aus einem Maschinenprogramm Quelltext erzeugt. So können Maschinenprogramme bequem verändert und dann neu assembliert werden.

REASSE

Der Reassembler (Listing 1) erzeugt aus einem Maschinenprogramm Quelltext, der mit Hypra-Ass editiert, verändert und wieder assembliert werden kann. Der Reassembler belegt den Speicherplatz von $C000 bis $CB00, kann aber mit dem SMON in jeden Bereich verschoben werden. Neben dem eigentlichen Reassembler stehen noch einige Basic-Befehle zur Verfügung, mit denen zum Beispiel Einsprungpunkte im Quelltext durch ein Label markiert werden können. Es läßt sich auch vorherbestimmen, ob der Reassembler selbständig nach Tabellen suchen soll oder nicht. Weiterhin läßt sich der Aufbau des Quelltextes in einigen Punkten mitbestimmen. Alle dazu nötigen Informationen werden dem Reassembler in einem kleinen Basic-Informationsprogramm mitgeteilt. Es stehen dafür drei neue Basic-Befehle zur Verfügung:

←P adresse: Mit diesem Befehl lassen sich Einsprungpunkte im Quelltext durch ein Label markieren. So sind Adressen, die mit SYS angesprungen werden, im Quelltext leichter auffindbar.

←T adresse, adresse: Mit diesem Befehl teilen Sie dem Reassembler die Lage von Tabellen mit. Die erste Adresse zeigt auf das erste und die zweite Adresse auf das letzte Byte der Tabelle. Tabellen werden vom Reassembler nicht reassembliert, sondern erscheinen im Quelltext in Form eines Hex-Dumps (siehe Bild 1; Zeile 190 und Bild 2 Zeile 230 bis 310).

←E (byte): Der »E«-Befehl startet den Reassembler und steht am Ende des Informationsprogramms. Es wird nun aus einem Maschinenprogramm ein Quelltext erzeugt, der im Basic-Speicher abgelegt und anschließend wie ein normales Basic-Programm gespeichert oder editiert und mit Hypra-Ass assembliert werden kann.

Der Aufbau des Quelltextes läßt sich geringfügig beeinflussen, indem hinter den »E«-Befehl eine Zahl zwischen 0 und 255 eingegeben wird. Bei dieser Zahl handelt es sich um ein sogenanntes Informations-Byte. Die einzelnen Bits dieses Informations-Bytes haben folgende Bedeutung:

Informations-Byte:	o	o	o	o	o	o	o	o
Bit:	7	6	5	4	3	2	1	0
Wertigkeit:	128	064	032	016	008	004	002	001

Um zum Beispiel Bit 1 und Bit 6 auf 1 zu setzen, sind die Wertigkeiten der einzelnen Bits zu addieren. In diesem Fall 2 + 64 = 66

Bit 0 gesetzt: Alle Zeropage-Adressen (Adressen von $00 bis $FF) werden durch ein Label mit nur drei Buchstaben (normal fünf) markiert.

Bit 1 gesetzt: Nach den Befehlen RTS, RTI, BRK und JMP wird eine Kommentarzeile in den Quelltext eingefügt (Zeile 220 in Bild 2). Dadurch wird der Quelltext übersichtlicher.

Bit 2 gesetzt: Bei allen Befehlen mit unmittelbarer Adressierung (zum Beispiel LDA # $41) wird der Operand zusätzlich im ASCII-Format ausgegeben (LDY #$00 ;" . " — Zeile 140 und 160 in Bild 2), vorausgesetzt, er liegt zwischen 32 und 96 oder zwischen 160 und 224. Für den Fall, daß er außerhalb dieses Zahlenbereichs liegt, wird nur ein Punkt ausgegeben.

Bit 3 gesetzt: Zwischen je zwei Tabellenzeilen wird eine Kommentarzeile eingefügt (Zeile 240, 260, 280, 300 in Bild 2). Dieses erhöht die Übersichtlichkeit.

Bit 4 gesetzt: Der ASCII-Ausdruck wird bei Tabellen unterdrückt.

Bit 5 gesetzt: Ist dieses Bit gesetzt, werden externe Label und Tabellenlabel speziell gekennzeichnet (Zeile 100 und 230 in Bild 2). Tabellen wird ein »T« vorangestellt (zum Beispiel TLC000) und externen Label (Label die außerhalb des zu reassemblierenden Bereichs liegen) ein »E« (zum Beispiel ELC000).

Bit 6 gesetzt: Ist das Bit 6 gesetzt, sucht der Reassembler selbständig nach Tabellen. Es wird kein Quelltext, sondern ein Basic-Informationsprogramm generiert, das die Start- und Endadressen aller gefundenen Tabellen enthält. Dieses kann mit LIST oder — wenn Hypra-Ass geladen wurde — mit /E geLISTet und geändert werden.

Bit 7 gesetzt: Der Reassembler reassembliert die Speicherinhalte, die sich unter dem ROM im RAM befinden. Dadurch ist es möglich, Programme zu reassemblieren, die sich unter dem Basic-Interpreter oder Betriebssystem befinden.

Aus den drei neuen Basic-Befehlen setzt sich jedes Informationsprogramm zusammen. Es wird mit folgender Befehls-Sequenz im Direktmodus gestartet:
```
SYS 49152, anfadr, endadr+1:RUN
```
anfadr = Anfangsadresse des Maschinenprogramms, das reassembliert werden soll.
endadr = Endadresse des Maschinenprogramms, das reassembliert werden soll.

Um zum Beispiel den Reassembler durch sich selbst reassemblieren zu lassen, gehen Sie wie folgt vor:

1. Reassembler laden mit
```
LOAD"REASS",8,1
```
2. NEW <RETURN> eingeben
3. Folgendes Basic-Informationsprogramm eingeben:
```
20←P $C000 ;kennzeichnet die Adresse $C000 durch
        ein Label
30←T $C813,$CAFF ;definiert eine Tabelle im
        Bereich von $C813 bis $CAFF
40←E 15 ;Startet den Reassembler und setzt die
        Bits 0 bis 3
```
4. Direktmodus eingeben.
```
SYS 49152,$C000,$CB00:RUN <RETURN>
```
Die SYS-Zeile, mit der das Informationsprogramm gestartet wird, teilt dem Reassembler mit, daß das zu reassemblierende Maschinenprogramm im Bereich von $C000

MBLER
zu Hypra-Ass

bis $CAFF ($CB00 - 1) liegt. In Zeile 20 trifft der Reassembler auf den »P«-Befehl, der dazu auffordert, die Adresse $C000 durch ein Label zu markieren. Der »T«-Befehl in Zeile 30 definiert eine Tabelle im Bereich $C813 bis $CAFF und der »E«-Befehl in Zeile 40 startet den Reassembler.

In weniger als 8 Sekunden wird nun ein etwa 17 KByte langer Quelltext erzeugt, der mit LIST oder — wenn Hypra-Ass geladen und gestartet wurde — mit dem /E-Befehl geLISTet und mit RUN assembliert werden kann.

Wie Sie sicherlich schon bemerkt haben, verarbeitet der Reassembler nicht nur Dezimal-, sondern auch Hexadezimalzahlen. Eine Hexadezimalzahl beginnt mit einem Dollar-Zeichen ($), dem genau vier Hex-Ziffern folgen müssen. Beispiel:
$0073, $C000, $FFFF

Besonderheiten

1. Der Reassembler arbeitet ausgezeichnet mit Hypra-Ass zusammen. Dabei ist es jedoch übersichtlicher, das Informationsprogramm ohne Leerzeichen einzugeben, weil Hypra-Ass nach dem ersten Leerzeichen einen Tabulator einfügt. Das Aussehen des Quelltextes würde dadurch verunstaltet. Gestartet wird das Informationsprogramm wie gewohnt mit RUN. (Vergessen Sie nicht den Minusstrich vor jeder Zeile, wenn Hypra-Ass geladen ist.)

2. Aus programmtechnischen Gründen kann es vorkommen, daß der Reassembler im ersten Pass ein Maschinenprogramm anders reassembliert als im zweiten. Dadurch können in Pass 2 Sprungadressen im Maschinenprogramm auftauchen, die in Pass 1 nicht gefunden wurden und deshalb auch im Quelltext nicht durch ein Label markiert werden. Der Reassembler ersetzt in diesem Fall die Sprungadresse nicht durch ein Label, sondern stellt sie als Hex-Zahl im Quelltext dar. An die entsprechende Zeile werden 3 Fragezeichen angehängt.

3. 3-Byte-Befehle, die bei der Assemblierung als 2-Byte-Befehle interpretiert werden (BIT $A9 $00 = .BY $2C LDA #$00), werden nicht reassembliert. Statt dessen werden die 3 Byte mit vorangestelltem .BY-Pseudoopcode in den Quelltext eingefügt. Der reassemblierte Befehl wird aber als Kommentar an die entsprechende Zeile angefügt.

4. Es ist möglich, ein Programm so zu reassemblieren, als ob es in einem anderen Bereich läge. Dazu ist an den SYS-Befehl eine weitere Adresse anzuhängen:
SYS 49152, anfadr, endadr, get
anfadr und endadr geben die Anfangs- und Endadresse des Bereichs an, in dem das Maschinenprogramm liegen soll. get gibt die Anfangsadresse des Bereichs an, in dem das Programm tatsächlich liegt.

So kann man zum Beispiel die Kopie der CHRGET-Routine ab $E3A2 reassemblieren, als ob sie im Bereich von $0073 bis $008A liegen würde. Dazu ist im Direktmodus folgende Zeile einzugeben:
```
SYS 49152, $0073, $008A, $E3A2:←E
```
Da keine Tabellen in diesem Bereich liegen, kann auf ein Informationsprogramm verzichtet werden.

5. Es ist möglich, während der Reassemblierung den erzeugten Quelltext auf Diskette zu schreiben. Dadurch bleibt der Basic-Speicher für andere Programme frei. Dazu ist vor dem SYS-Befehl, mit dem die Start- und Endadresse übergeben wird, ein Programmfile zu öffnen. Mit dem Befehl CMD wird die Ausgabe auf das entsprechende Gerät umgeleitet. Das könnte wie folgt aussehen:
```
OPEN 1,8,1,"NAME,P,W":CMD 1:SYS 49152,$C000,
$CB00:←E 64
```
Mit dem OPEN-Befehl wird ein Programmfile mit dem Namen »Name« zum Schreiben geöffnet. Der CMD-Befehl leitet die Ausgabe auf das Gerät mit der Gerätenummer 8 um (Disketten-Laufwerk). Der SYS-Befehl startet schließlich den Reassembler, dem durch den »E«-Befehl noch mitgeteilt wird, daß kein Quelltext, sondern ein Informationsprogramm erstellt werden soll. Das Informationsprogramm wird unter dem Namen »Name« auf Diskette gespeichert.

Vorsicht! Dieser Programmteil ist nicht gegen Fehlbedienung abgesichert. So führt eine nicht eingelegte Diskette zum Absturz des Systems. In einem solchen Fall ist ein RESET auszulösen. Hypra-Ass kann anschließend mit SYS 2168 neu gestartet werden. Außerdem sollte nach jedem Speichern die RUN/STOP-RESTORE-Taste gedrückt werden.

Fehlermeldungen

SYNTAX ERROR: Ein Basic-Befehl wurde falsch eingegeben oder eine Hex-Zahl besteht aus weniger als 4 Hex-Ziffern.

OUT OF MEMORY: Es steht zu wenig Speicherplatz für den Quelltext zur Verfügung oder im Maschinenprogramm kommen mehr als 2700 verschiedene Label vor.

ILLEGAL QUANTITY: Vor einer Hex-Zahl fehlt das Dollar-Zeichen ($) oder das Tabellenende liegt vor dem Tabellenanfang oder die Tabellen überschneiden sich oder die angegebene Adresse liegt nicht im Maschinenprogramm.

TYPE MISMATCH: In einer Hex-Zahl stehen falsche Hex-Ziffern. Die Adresse, die schon als Einsprungpunkt markiert wurde, darf nicht als Tabellenanfang oder -ende angegeben werden. Im Informationsprogramm darf keine Adresse doppelt vorkommen.

```
hypra-ass  assemblerlisting:

             10   -.li 1,4,7
;*****************************
;*   Ausgabe eines Textes auf  *
;*         dem Bildschirm      *
;*****************************
             60   -.ba $8000 ;Startadresse = $8000
;
             80   -.eq ausgabe = $ffd2 ;Dies ist ein externes Label
;
8000 a000   :100  -anfang    ldy #0        ;Schleifenzaehler auf 0 setzen
8002 b91080 :110  -loop      lda text,y    ;Zeichen holen
8005 c923   :120  -          cmp #"#"      ;mit Endekennzeichen vergleichen
8007 f006   :130  -          beq ende
8009 20d2ff :140  -          jsr ausgabe   ;Zeichen ausgeben
800c c8     :150  -          iny           ;Schleifenzaehler + 1
800d d0f3   :160  -          bne loop
800f 60     :170  -ende      rts           ;Ende und zurueck ins Basic
;
             190  -text      .tx "Dies ist ein Beispieltext#"
```

◀ **Bild 1. Beispielprogramm zum »Reassembler« (Original Quelltext erstellt mit Hypra-Ass)**

```
100  -          .eq elffd2=$ffd2
110  -;
120  -          .ba $8000
130  -;
140  -18000     ldy #$00          ; "."
150  -18002     lda t18010,y
160  -          cmp #$23          ; "#"
170  -          beq 1800f
180  -          jsr elffd2
190  -          iny
200  -          bne 18002
210  -1800f     rts
220  -;
230  -t18010    .by $c4,$49,$45,$53,$20,$49,$53,$54; "Dies ist"
240  -;
250  -          .by $20,$45,$49,$4e,$20,$c2,$45,$49; " ein Bei"
260  -;
270  -          .by $53,$50,$49,$45,$4c,$54,$45,$58; "spieltex"
280  -;
290  -          .by $54           ; "t"
300  -;
310  -t18029    .by $23           ; "#"
```

▲ **Bild 2. Reassembler-Quelltext zum Beispielprogramm aus Bild 1**

Verschieben des Reassemblers

Der Reassembler benutzt in der Zeropage verschiedene Speicherzellen als Kurzzeitspeicher. Der Langzeitspeicher dagegen liegt unter dem Betriebssystem ($E000 bis $FFFF). Dort befindet sich auch ab Adresse $E028 die Label-Tabelle. In den Langzeitspeicher sollte nicht hineingePOKEt werden.

Der Reassembler kann mit SMON ohne Schwierigkeiten im Speicher verschoben werden. Um den Reassembler nach $9000 zu verschieben, sind folgende SMON-Befehle einzugeben:

```
W C000 CB00 9000
V C000 CB00 9000 9000 9813
```

Beispiel zu den Basic-Erweiterungen

Laden Sie Hypra-Ass, starten Sie ihn und laden anschließend den Reassembler. Geben Sie NEW und danach im Direktmodus

```
SYS 49152,$1000,$1FD7:←E 64 <RETURN>
```

ein. Der Reassembler bekommt durch den SYS-Befehl die Start- und Endadresse des Maschinenprogramms mitgeteilt. Der »E«-Befehl setzt Bit 6 des Informations-Bytes und startet den Reassembler. Das gesetzte Bit 6 bewirkt, daß kein Quelltext, sondern ein Informationsprogramm erstellt wird. LISTen Sie das Programm mit /E. Sie sehen eine Reihe von »T«-Befehlen und zum Schluß einen »E«-Befehl. Schreiben Sie hinter diesen Befehl die Zahl 32, drücken die RETURN-Taste und geben folgende Zeile im Direktmodus ein:

```
OPEN 1,8,1,"REASS DEMO,P,W":CMD 1:SYS
49152,$1000,$1FD7:GOTO 100 <RETURN>
```

Mit dem OPEN-Befehl wird ein Programmfile mit dem Namen »REASS DEMO« zum Schreiben geöffnet. Der nachfolgende CMD-Befehl leitet die Ausgabe des Quelltextes auf dieses File um. Durch den SYS-Befehl wird dem Reassembler die Start- und Endadresse des Maschinenprogramms mitgeteilt. Der GOTO-Befehl startet schließlich das Informationsprogramm (der RUN-Befehl darf dazu nicht benutzt werden, da er das geöffnete File schließen würde). Die »T«-Befehle im Informationsprogramm werden ausgeführt, bis der »E«-Befehl Bit 5 setzt und den Reassembler startet. Das gesetzte fünfte Bit bewirkt, daß externe Label und Tabellenlabel speziell gekennzeichnet werden. Das so auf Diskette erzeugte Programmfile kann mit LOAD "REASS DEMO",8 geladen, geLISTet, editiert und assembliert werden.

(Martin Wehner/sk)

```
Name : reass                c000 cb01
-------------------------------------

c000 : 20 4e c1 20 53 e4 ad 05  d9
c008 : c8 8d 02 03 ad 06 c8 8d  c1
c010 : 03 03 ad ff c7 8d 0a 03  17
c018 : ad 00 c8 8d 0b 03 20 53  99
c020 : c7 8d 09 e0 8c 08 e0 8d  b4
c028 : 01 e0 8c 00 e0 8d 03 e0  05
c030 : 8c 02 e0 48 98 48 20 53  f1
c038 : c7 8d 05 e0 8c 04 e0 20  d0
c040 : 79 00 f0 09 20 53 c7 8d  ed
c048 : 01 e0 8c 00 e0 68 85 14  6c
c050 : 68 85 15 a9 ff a2 02 9d  4e
c058 : 28 e0 ca 10 fa ad 02 c8  5c
c060 : 8d 08 03 ad 03 c8 8d 09  27
c068 : 03 20 d1 c0 20 d7 c0 4c  64
c070 : 1e c1 a9 00 85 0d 20 73  01
c078 : 00 b0 03 4c f3 bc 20 13  e6
c080 : b1 90 03 4c 28 af c9 24  33
c088 : f0 03 4c 9a ae 20 a0 c0  50
c090 : 85 62 20 a0 c0 85 63 a2  6d
c098 : 90 38 20 49 bc 4c 73 00  72
c0a0 : 20 bf c0 20 b5 c0 0a 0a  72
c0a8 : 0a 0a 85 14 20 bf c0 20  de
c0b0 : b5 c0 05 14 60 c9 3a 90  e8
c0b8 : 02 69 08 29 0f 38 60 20  0b
c0c0 : 73 00 90 12 c9 41 90 04  8b
c0c8 : c9 47 90 0a a2 16 6c 00  27
c0d0 : 03 78 a9 35 85 01 60 a0  43
c0d8 : 00 20 df c0 90 df c8 a5  6f
c0e0 : 14 d9 02 e0 c8 a5 15 f9  7f
c0e8 : 02 e0 60 20 73 00 20 56  db
c0f0 : c7 a2 80 20 36 c1 20 53  c5
c0f8 : c7 20 d1 c0 20 3c c5 b0  b8
c100 : 1d 20 09 c1 a2 00 4c 36  70
c108 : c1 20 57 c6 a0 00 b1 59  0b
c110 : f0 14 10 f5 c8 a5 14 d1  7b
c118 : 59 c8 a5 15 f1 59 b0 06  9a
c120 : a9 37 85 01 58 60 a2 0f  18
c128 : 20 20 c1 6c 00 03 20 73  d6
c130 : 00 20 56 c7 a2 02 20 69  5c
c138 : c0 20 d1 c0 20 27 c5 20  27
c140 : cd c5 d0 e4 a1 59 30 d8  18
c148 : 20 20 c1 4c ea a7 a0 0b  f7
c150 : 20 d1 c0 b9 00 03 48 b9  6d
c158 : 10 e0 99 00 03 68 99 10  39
c160 : e0 88 10 ef 4c 20 c1 20  93
c168 : 4e c1 6c 02 03 20 cd c1  de
c170 : 20 bc c1 b0 f8 a0 00 8c  22
c178 : af 02 2c a8 02 30 03 20  36
```

```
c180 : 20 c1 b1 5d aa 20 d1 c0  0d
c188 : e6 5d d0 02 e6 5e e6 5f  4d
c190 : d0 02 e6 60 a5 5f 85 14  bb
c198 : 29 3f d0 11 20 20 c1 a5  0c
c1a0 : 9a c9 08 b0 05 a9 c0 20  18
c1a8 : d2 ff 20 d1 c0 a5 60 85  82
c1b0 : 15 20 d7 c0 90 06 a9 80  c4
c1b8 : 8d af 02 60 a0 01 a5 5f  11
c1c0 : d1 fd c8 a5 60 f1 fd 90  25
c1c8 : 03 ee af 02 60 18 a5 fd  c8
c1d0 : 69 03 85 fd 90 02 e6 fe  8f
c1d8 : 60 20 70 c1 8e a9 02 bd  56
c1e0 : ff c8 85 5c bd ff c9 8d  4e
c1e8 : b0 02 f0 17 b0 4f 30 4d  99
c1f0 : 29 0f 4a b0 1e 4a b0 0c  59
c1f8 : 20 70 c1 8e aa 02 a2 00  d8
c200 : 8e ab 02 60 20 70 c1 8e  9a
c208 : aa 02 b0 32 20 70 c1 8e  cf
c210 : ab 02 60 20 70 c1 8a 30  78
c218 : 0e 18 65 5f 8d aa 02 a5  f9
c220 : 60 69 00 8d ab 02 60 49  c5
c228 : ff 85 5b e6 5b 38 a5 5f  6a
c230 : e5 5b 8d aa 02 a5 60 e9  1e
c238 : 00 8d ab 02 60 18 a9 81  9a
c240 : 69 00 8d b0 02 a9 02 85  a3
c248 : 5c 60 20 73 00 f0 04 c9  76
c250 : 5f f0 03 4c e7 a7 20 73  95
c258 : 00 c9 50 d0 03 4c 2e c1  3a
c260 : c9 54 d0 03 4c eb c0 c9  a3
c268 : 45 f0 05 a2 0b 6c 00 03  d5
c270 : a2 00 8e ae 02 20 73 00  7b
c278 : f0 03 20 9e b7 8e a8 02  5c
c280 : 20 d1 c0 20 3e c7 20 27  ae
c288 : c5 20 bc c1 b0 1f 20 d9  fd
c290 : c1 ad b0 02 30 10 29 0f  db
c298 : f0 0c c9 08 f0 08 a2 01  de
c2a0 : 20 bf c5 20 ca c5 ad af  06
c2a8 : 02 f0 e3 30 17 a0 00 b1  fb
c2b0 : fd 10 db 20 70 c1 90 fb  ff
c2b8 : a0 00 b1 fd d0 f5 20 70  a2
c2c0 : c1 4c a6 c2 20 f5 c7 20  ba
c2c8 : 3e c7 20 d1 c7 a5 2b 85  8d
c2d0 : fb a5 2c 85 fc 20 75 c7  90
c2d8 : a0 00 b1 fd c9 ff f0 35  6f
c2e0 : 20 2f c7 20 d7 c0 90 27  a2
c2e8 : 20 1f c6 20 78 c7 a2 00  9e
c2f0 : 20 a4 c6 ad a8 02 29 20  49
c2f8 : f0 05 a9 45 20 34 c6 20  7d
c300 : b6 c6 a9 3d 20 34 c6 a9  3d
c308 : 24 20 b8 c6 20 70 c7 20  28
c310 : cd c1 4c d8 c2 20 3e c7  a2
c318 : 20 6d c7 20 1f c6 20 78  7e
c320 : c7 a2 01 20 a4 c6 a5 5f  53
c328 : 8d aa 02 a5 60 8d ab 02  65
c330 : a9 24 20 b8 c6 20 70 c7  c9
c338 : 20 6d c7 a0 00 b1 fd 30  fb
c340 : 10 20 2f c7 a0 00 20 df  6f
c348 : c0 b0 06 20 cd c1 4c 3b  78
c350 : c3 20 bc c1 90 03 4c 91  00
c358 : c4 20 1f c6 20 d9 c1 ad  00
c360 : ae 02 d0 3a ad b0 02 c9  87
c368 : 81 d0 33 ad af 02 d0 2e  7e
c370 : a8 a9 80 91 57 c8 8c ae  8a
c378 : 02 ad 08 e0 91 57 c8 ad  c1
c380 : 09 e0 91 57 20 6f c6 b1  45
c388 : fd 8d 09 e0 91 57 88 b1  04
c390 : fd 8d 08 e0 91 57 88 a9  bb
c398 : 00 91 57 20 62 c5 20 78  00
c3a0 : c7 ad b0 02 30 1c 29 0f  51
c3a8 : f0 69 c9 06 d0 12 ad ab  2c
c3b0 : 02 d0 0f a2 02 20 a4 c6  74
c3b8 : a9 83 4c 70 c4 4c 53 c4  c9
c3c0 : b0 fb 20 a2 c6 ad b0 02  6b
c3c8 : 30 f0 29 10 f0 05 a9 28  eb
c3d0 : 20 34 c6 20 bf c5 20 75  56
c3d8 : c5 20 b8 c6 ad b0 02 29  6f
c3e0 : 70 f0 33 c9 10 d0 08 a9  c9
c3e8 : 29 20 34 c6 4c 16 c4 c9  23
c3f0 : 40 90 11 f0 05 a9 29 20  5d
c3f8 : 34 c6 a9 2c a2 59 20 30  55
c400 : c6 4c 16 c4 a9 2c a2 58  42
c408 : 20 30 c6 ad b0 02 29 10  87
c410 : 4c e1 c3 20 a2 c6 a5 5c  f1
c418 : c9 37 d0 06 20 75 c7 20  7f
c420 : a2 c6 ad af 02 30 2a 20  11
c428 : 70 c7 a5 5c c9 38 90 19  44
c430 : ad ae 02 d0 0a a5 5f 8d  35
c438 : 08 e0 a5 60 8d 09 e0 ad  26
c440 : a8 02 29 02 f0 03 20 6d  f6
c448 : c7 ad af 02 d0 43 4c 59  1d
c450 : c3 30 58 20 a2 c6 a9 23  93
c458 : 20 34 c6 ad aa 02 8d a9  3e
c460 : 02 a2 71 ad a8 02 29 04  0d
c468 : f0 02 a2 81 8a 8d b0 02  0e
c470 : 29 03 85 5b aa bd a8 02  27
c478 : 9d 1f e0 ca d0 f7 20 ec  5d
c480 : c6 ad b0 02 c9 70 b0 8e  89
c488 : 20 75 c7 20 a2 c6 4c d3  92
c490 : c3 20 2f c7 20 1f c6 20  7e
c498 : 63 c6 a0 00 8c ae 02 b1  30
c4a0 : fd f0 70 30 2e 20 b6 c6  84
c4a8 : 4c 5c c3 a2 00 a9 ff 81  b8
c4b0 : 57 ad ae 02 f0 03 20 4a  06
c4b8 : c5 20 7d c7 20 33 c6 20  dd
c4c0 : 30 c6 20 20 c1 a5 b8 20  cc
c4c8 : c3 ff 20 b5 ab 6a 85 9d  a9
c4d0 : 4c ab e1 20 e6 c7 20 3d  16
c4d8 : c2 ad a8 02 29 10 f0 05  bc
c4e0 : a9 71 8d b0 02 a0 00 84  e9
c4e8 : 5b 20 70 c1 a4 5b 8a 99  2a
c4f0 : 20 e0 e6 5b b0 0a c0 07  12
c4f8 : d0 ef 20 16 c6 4c d6 c4  3e
c500 : a0 00 b1 fd f0 04 09 e0  e1
c508 : 91 fd 20 e6 c6 20 5c c7  eb
c510 : 4c 16 c4 20 e6 c7 20 70  aa
c518 : c1 8e a9 02 20 3d c2 4c  5b
c520 : 9e c3 20 3c c5 90 10 ad  ac
c528 : 08 c8 85 59 ad 09 c8 85  72
c530 : 5a 4c 37 c5 20 57 c6 20  4f
c538 : 3c c5 90 f8 a0 02 b1 59  2e
c540 : c5 15 d0 05 88 b1 59 c5  6c
c548 : 14 60 38 a5 57 e9 03 85  2b
c550 : 57 b0 02 c6 58 a0 01 ad  43
c558 : 04 e0 91 57 c8 ad 05 e0  eb
c560 : 91 57 38 a0 01 b1 57 e9  8e
c568 : 01 91 57 c8 b1 57 e9 00  9e
c570 : 91 57 4c 6f c6 20 22 c5  2f
c578 : d0 08 a0 00 b1 59 c9 ff  81
c580 : 90 07 a9 37 85 5c a9 24  0f
c588 : 60 ad a8 02 29 20 f0 1c  b9
c590 : 20 d7 c0 90 1a a9 45 20  22
c598 : 34 c6 4c ac c5 20 57 c6  20
c5a0 : b1 59 f0 13 c9 ff f0 04  05
c5a8 : c9 80 d0 f1 a9 4c 60 a0  e3
c5b0 : 00 b1 59 c9 80 d0 e9 a9  a2
c5b8 : 54 20 34 c6 4c ac c5 ad  9f
c5c0 : aa 02 85 14 ad ab 02 85  9a
c5c8 : 15 60 20 22 c5 f0 44 86  5c
c5d0 : 61 a5 59 85 5b a5 5a 85  62
c5d8 : 5c a0 02 b1 5b b6 61 99  5f
c5e0 : 61 00 8a 91 5b 88 10 f3  38
c5e8 : 18 a5 5b 69 03 85 5b 90  c2
c5f0 : 02 e6 5c e0 ff d0 e2 a0  ec
c5f8 : 01 a5 14 91 59 c8 a5 15  a0
c600 : 91 59 a5 5c cd 0c c8 90  b4
c608 : 07 a5 5b cd 0b c8 b0 7d  27
c610 : a2 00 60 a2 0e 60 20 e6  51
c618 : c6 20 5c c7 20 70 c7 18  d3
c620 : ad ac 02 69 0a 8d ac 02  95
c628 : 90 03 ee ad 02 ae ad 02  fb
c630 : 20 34 c6 8a 2c a8 02 70  5e
c638 : 66 a4 9a c0 08 b0 53 a0  44
c640 : 00 91 fb e6 fb a4 fb 84  c2
c648 : 2d d0 02 e6 fc a4 fc 84  2d
c650 : 2e c4 38 b0 38 90 48 18  5e
c658 : a5 59 69 03 85 59 90 02  ce
c660 : e6 5a 60 a0 02 b1 fd 91  68
c668 : 57 88 10 f9 4a b0 f3 2c  99
c670 : a8 02 50 0b 18 a5 57 69  6e
c678 : 03 85 57 90 02 e6 58 a5  2a
c680 : 58 cd 12 c8 90 dc a5 57  92
c688 : cd 11 c8 90 d5 a2 10 4c  6d
c690 : 28 c1 48 a8 20 20 c1 98  fb
c698 : 20 d2 ff 20 d1 c0 68 a0  2b
c6a0 : 00 60 a6 5c bd 4b c8 20  9f
c6a8 : 34 c6 bd 87 c8 20 34 c6  8b
c6b0 : bd c3 c8 4c 34 c6 a9 4c  c3
c6b8 : 20 34 c6 ad a8 02 29 01  9b
c6c0 : f0 05 ad ab 02 f0 06 ad  2f
c6c8 : ab 02 20 d0 c6 ad aa 02  1f
c6d0 : 48 4a 4a 4a 4a 20 db c6  bc
c6d8 : 68 29 0f c9 0a 90 02 69  d2
c6e0 : 06 69 30 4c 34 c6 20 78  1b
c6e8 : c7 20 a2 c6 a2 00 f0 05  39
c6f0 : a9 2c 20 34 c6 a9 24 20  c8
c6f8 : 34 c6 bd 20 e0 20 d0 c6  e3
c700 : e8 e4 5b 90 eb ad b0 02  36
c708 : 10 33 20 75 c7 20 2b c7  22
c710 : a2 00 bd 20 e0 a8 29 7f  1d
c718 : c9 20 90 04 c9 60 90 02  7c
c720 : a0 2e 98 20 34 c6 e8 e4  e8
c728 : 5b 90 e7 a9 22 d0 4b a0  11
c730 : 00 20 34 c7 c8 b1 fd 99  8c
c738 : 13 00 99 a9 02 60 ad 08  d1
c740 : c8 85 fd ad 09 c8 85 fe  eb
c748 : a2 04 bd ff df 95 5c ca  0d
c750 : d0 f8 60 20 fd ae 20 8a  a3
c758 : ad 4c f7 b7 ad a8 02 29  9b
c760 : 08 f0 da 20 70 c7 20 1f  9f
c768 : c6 a9 3b d0 0d 20 66 c7  e7
c770 : a9 00 20 34 c6 20 69 c7  4b
c778 : a9 20 4c 34 c6 2c a8 02  3f
c780 : 8e a8 02 50 60 20 d1 c7  cb
c788 : 20 f5 c7 20 75 c7 20 1f  ed
c790 : c6 a9 5f 20 34 c6 b1 57  f6
c798 : c9 ff f0 30 c9 80 f0 0c  1f
c7a0 : a9 50 20 b5 c7 98 20 34  5a
c7a8 : c6 4c 8b c7 a9 54 20 b5  99
c7b0 : c7 a9 2c d0 ed 20 34 c6  af
c7b8 : c8 b1 57 8d aa 02 c8 b1  22
c7c0 : 57 8d ab 02 a9 24 20 b8  b7
c7c8 : c6 4c 74 c6 a9 45 20 34  58
c7d0 : c6 a9 5a 8d ac 02 a2 00  18
c7d8 : 8e ad 02 ad 0e c8 85 57  5f
c7e0 : ad 0f c8 85 58 60 ad a8  88
c7e8 : 02 29 20 f0 05 a9 54 20  d4
c7f0 : 34 c6 4c b6 c6 a5 9a c9  09
c7f8 : 08 90 ea 4c 34 c6 2c 72  9c
c800 : c0 2c 4a c2 2c 67 c1 2c  1f
c808 : 28 e0 2c f0 f7 2c 00 f8  9c
c810 : 2c f0 ff 2e 2e 2e 20 48  df
c818 : 59 50 52 41 2d 52 45 2c  29
c820 : 20 50 55 42 4c 49 53 48  f3
c828 : 45 44 20 42 59 20 36 34  b8
```

Listing 1. »REASS« bitte mit dem MSE eingeben.

Fortsetzung auf Seite 162.

Hypra-Ass Reass, SMON Befehlsübersicht

Leistungsstarke Programme wie der SMON sind oft komplex in der Bedienung. Daher wird Ihnen die Befehlsübersicht bei der Arbeit mit diesen Programmen sehr behilflich sein.

Selbst für den eingefleischten Profi ist es fast unmöglich, alle Befehle von Hypra-Ass, Reassembler und SMON jeder Zeit parat zu haben. Um aber diese Programme optimal zu nutzen, ist die Kenntnis aller Befehle und deren Wirkungsweise sehr wichtig. Die folgenden Befehlsübersichten sollen Ihnen das ewige Nachschlagen ersparen. Natürlich kann es sich bei einer Übersicht nicht um eine ausführliche Anleitung handeln. Bei Verständnisschwierigkeiten finden Sie nähere Informationen an den entsprechenden Stellen in den Artikeln. (sk)

Quickreferenz Hypra-Ass

Editorbefehle von Hypra-Ass

/A 100, 10	Automatische Zeilennumerierung. (Startzeile, Schrittweite)
/O	RENEW eines Quelltextes.
/D 100-200	Löschen von Zeilen und Zeilenbereichen.
/E 100-200	Listen von Zeilen und Zeilenbereichen.
TO,13;T1,24;T2,0;T3,10	Setzen von Tabulatoren. T0 = Tabulator für Assemblerbefehle. T1 = Tabulator für den Kommentar. T2 = Anzahl Blanks am Anfang einer Ausgabezeile. T3 = Tabulator für Symboltabelle.
/X	Verlassen des Assemblers.
/P1,100,200	Setzen eines Arbeitsbereichs (Page).
/ziffer (n)	Formatiertes Listen der Page.
/N1,100,10	Neu durchnumerieren einer Page mit Startnummer und Schrittweite.

Quickreferenz Reassembler

P adresse	Einsprungspunkt durch Label markieren.
T adresse, adresse	Tabelle definieren.
E (byte)	Startet den Reassembler. Die einzelnen Bits des Bytes haben folgende Bedeutung:
Bit 0 gesetzt	Alle Zeropage-Adressen durch ein Label mit drei Buchstaben markieren.
Bit 1 gesetzt	Nach RTS, RTI, BRK, JMP Kommentarzeile einfügen.
Bit 2 gesetzt	Bei unmittelbarer Adressierung ASCII-Zeichen ausgeben.
Bit 3 gesetzt	Zwischen jede zweite Tabellenzeile Kommentarzeile einfügen.
Bit 4 gesetzt	Der ASCII-Ausdruck wird bei Tabellen unterdrückt.
Bit 5 gesetzt	Externe- und Tabellenlabel kennzeichnen.
Bit 6 gesetzt	Nach Tabellen suchen.
Bit 7 gesetzt	Speicherbereiche unter dem RAM reassemblieren.

ROCKUS



Quickreferenz Hypra-Ass

Editorbefehle von Hypra-Ass

/F1, "string"	Suchen einer Zeichenkette in einer Page.
/R1, "string1", "string2"	String 2 wird innerhalb einer Page durch String 1 ersetzt.
/U 9000	Setzen des Quelltextstartes.
/B	Anzeige der aktuellen Speicherkonfiguration.
/S"name";/L"name";/V"name";/M"name"	Kurzform der Befehle SAVE, LOAD, VERIFY, MERGE.
/G9	Geräteadresse des Floppy-Laufwerks auf 9 umstellen.
/I	Lesen des Inhaltsverzeichnisses.
/K	Lesen des Fehlerkanals.
/@	Übermittlung von Diskettenbefehlen.
/CHO	Setzen der Hintergrundfarbe.
/CRO	Setzen der Rahmenfarbe.
/!	Ausgabe der Symboltabelle (unsortiert).
/!!	Ausgabe der Symboltabelle (sortiert).

Pseudo-Opcodes von Hypra-Ass

.BA adresse	Definiert Startadresse des Maschinenprogramms.
.EQ label = wert	Weist einem Label einen Wert zu.
.GL label = wert	Weist einem globalen Label einen Wert zu.
.BY 1,2,"a"	Einfügen von Byte-Werten in den Quelltext.
.WO 1234,label	Einfügen von Adressen in den Quelltext.
.TX "text"	Einfügen von Textblöcken in den Quelltext.
.AP "file"	Verketten von Quelltexten.
.OB "file,p,w"	Senden des Objektcodes zur Floppy.
.EN	Schließen des Objektfiles.
.ON ausdruck,sprung	Bedingter Sprung, wenn Ausdruck wahr.
.GO sprung	Unbedingter Sprung.
.IF ausdruck	Fortführung der Assemblierung bei .EL, falls Ausdruck falsch. Ansonsten hinter .IF bis zu .EL oder .EI.
.EL	ELSE Alternative zu den Zeilen, die hinter .IF stehen.
.EI	Ende der IF-Konstruktion.
.CO var1,var2	Übergabe von Labeln und Quelltext an nachgeladene Teile.
.MA makro (par1,par2)	Makrodefinitionszeile.
.RT	Ende der Makrodefinition.
...makro (par1,par2)	Makroaufruf.
.LI lfn,dn,ba	Senden von formatierten Listings (entspricht OPEN-Befehl).
.SY lfn,dn,ba	Senden der formatierten Symboltabelle.
.ST	Beendet die Assemblierung.
.DP t0,t1,t2,t3	Setzt die Tabulatoren aus dem Quelltext heraus.

Quickreferenz SMON. Die Klammern dürfen nicht mit eingegeben werden. Die Werte in den Klammern können, aber müssen nicht eingegeben werden.

A 4000	Zeilenassembler Startadresse = $4000.
B 4000 4200	Erzeugt Basic-DATA-Zeilen im Bereich $4000 bis $41FF
C 4010 4200 4013 4000 4200	Verschieben eines Programmes mit Adreßumrechnung. Entspricht W- und V-Befehl.
D 4000 (4100)	Disassembliert den Bereich von $4000 bis $4100
F	findet Zeichenketten (F), absolute Adressen (FA), relative Sprünge (FR), Tabellen (FT), Zeropage-Adressen (FZ) und Immediate-Befehle (FI).
GO 4000	Startet Maschinenprogramm (ab $4000)
IO 1	Ein-/Ausgabegerät auf Datasette umstellen
K A000 (A100)	Im angegebenen Bereich nach ASCII-Zeichen suchen.
L"name" (4000)	Laden eines Programmes an die richtige (oder angegebene) Adresse
M 4000 (4100)	Gibt den Inhalt des angegebenen Speicherbereichs als Hex-Byte und ASCII-Zeichen aus.
O 4000 4100 12	Füllt den angegebenen Bereich mit $12.
PO 5	Setzt Drucker-Geräteadresse auf 5
R	Registerinhalte anzeigen
S"name" 4000 4500	Speichert ein Programm von $4000 bis $4FFF.
TW (4000)	Einzelschrittmodus. Mit »J« können Unterprogramme in Echtzeit ausgeführt werden.
TB 4010 (05)	Breakpoint setzen (nach dem 5. Durchlauf)
TQ 4000	Schnellschrittmodus. Springt beim Erreichen eines Breakpoints in die Registeranzeige.
TS 4000 4020	Arbeitet ein Programm ab $4000 in Echtzeit ab und springt beim Erreichen von $4020 in die Registeranzeige.
V 6000 6200 4000 4100 4200	Ändert alle absoluten Adressen $4000 bis $41FF, die sich auf den Bereich $6000 bis $6200 beziehen, auf den neuen Bereich $4000.
W 4000 4300 5000	Verschiebt den Speicherinhalt von $4000 bis $42FF nach $5000.
X	Monitor verlassen.
# 49152	Dezimal umrechnen
$ 002B	Vierstellige Hex-Zahl umrechnen.
% 01100100	Achtstellige Binärzahl umrechnen.
? 0344+5234	Addition oder Subtraktion zweier vierstelliger Hex-Zahlen.
= 4000 5000	Vergleicht den Speicherinhalt von $4000 bis $5000.
Z	Ruft den Diskettenmonitor auf (falls implementiert). Dieser verfügt über folgende Befehle
R (12 01)	Liest Track $12 Sektor $01. Fehlt die Angabe hinter »R«, wird der logisch nächste Sektor gelesen.
W (12 01)	Schreibt Track $12 Sektor $01 auf Diskette. Fehlt die Angabe hinter »W«, werden die letzten Angaben von »R« benutzt.
M	Zeigt den Pufferinhalt als Hex-Dump.
X	Rücksprung zum Monitor.
F	Weitere Diskettenbefehle initialisieren (falls implementiert). Sind die Befehle initialisiert, stehen folgende Befehle zur Verfügung.
M (07)	Memory-Dump (Floppy-RAM/ROM) ausgeben.
V 6000 0400	Verschiebt einen 256-Byte-Block von $6000 ins Floppy-RAM nach $400.
@	Normale Diskettenbefehle senden
X	Zurück in normalen Diskettenmonitor.

Ist die Erweiterung »Neues vom SMON« implementiert, stehen folgende Befehle zur Verfügung:

Z 4000 (4100)	Gibt den Speicherinhalt von $4000 bis $40FF binär aus (ein Byte pro Zeile).
H 4000 (4100)	Gibt den Speicherbereich von $4000 bis $40FF binär aus (drei Byte pro Zeile).
N 4000 (4100)	Gibt den Speicherinhalt von $4000 bis $40FF im Bildschirmcode aus (32 Zeichen pro Zeile).
U 4000 (4100)	Wie »N« aber 40 Zeichen pro Zeile. Änderungen sind nicht möglich.
E 4000 (4100)	Füllt den Speicherbereich von $4000 bis $40FF mit $00.
Y 40	Verschiebt den SMON nach $4000.
Q 2000	Kopiert den Zeichensatz nach $2000.
J	Bringt letzten Ausgabebefehl zurück.

ROM-Routin

Das Rad ist schon erfunden! Ähnlich verhält es sich mit verschiedenen Routinen, die ein Assembler-Programmierer immer wieder benötigt. Aber warum soll man sich die Arbeit des Programmierens machen, wenn das Betriebssystem viele ständig benötigte Routinen schon enthält und man nur noch zu wissen braucht, ab welcher Adresse sie stehen?

Angenommen, Sie möchten in Assembler einige komplexe Dinge programmieren wie beispielsweise eine neue mathematische Funktion (wie wäre es mit dem Kotangens) und diese auf dem Bildschirm ausgeben. Das ist eine große Aufgabe, zu der zunächst einmal die Übernahme des Arguments in das Maschinenprogramm, dann einige Fließkomma-Rechenoperationen und schließlich die Ausgabe auf dem Bildschirm geschrieben werden müßten, wenn da nicht schon fast alles an verborgener Stelle als fertige Programm-Module im Computer vorhanden wäre!

Sowohl im unteren (von $A000 bis $BFFF) als auch im oberen ROM-Bereich (von $E000 bis $FFFF) liegt die Firmware fest verschachtelt vor. Der untere ROM-Abschnitt wird manchmal auch Basic-Interpreter, der obere ROM-Bereich Betriebssystem genannt, wobei diese Einteilung aber den Kern der Sache nicht genau trifft, denn Interpreter, Editor und Betriebssystem führen ein gemischtes Dasein quer durch alle genannten ROM-Bereiche hindurch.

Mindestens fünf Informationen braucht ein Assembler-Programmierer, wenn er das breite Programmangebot des ROMs nutzen möchte:

1. Einsprungadresse
2. Format der Eingabeparameter
3. Adressen der Eingabeparameter
4. Adressen der Ausgabeparameter
5. Format der Ausgabeparameter

Nicht alle Routinen, die man benutzen kann, erfordern alle fünf Informationen, manche weniger, einige auch mehr und schließlich gibt es auch Programmroutinen, die noch den Aufruf einer oder sogar mehrerer anderer Routinen nötig machen.

In den beigefügten Tabellen sind - nach Anwendungen sortiert - die wichtigsten Firmware-Möglichkeiten mit den erforderlichen Ein- und Ausgabeparametern aufgeführt. Das sind natürlich beileibe nicht alle. Die Auswahl erfolgte subjektiv! Es sind einfach diejenigen, die mir bislang am häufigsten untergekommen sind. Außerdem wurde auf die Kernel-Routinen verzichtet: Man findet diese sehr gut dokumentiert bereits in einer Reihe von Büchern und im Assembler-Kurs.

Wichtige Adressen

Die Tabellen nennen den Label-Namen, die Einsprungadresse und geben eine Kurzbeschreibung der Funktionen. Das Ein- und auch das Ausgabeformat ist ebenso angegeben wie auch die Adressen, an denen diese Parameter übergeben werden. Die verwendeten Bezeichnungen halten sich eng an die im Assembler-Kurs kennengelernten. Sie sind allgemein üblich:

FAC	Fließkomma-Akku 1
ARG	Fließkomma-Akku 2
A	Akkumulator
X,Y	X-, Y-Register
A/Y	2-Byte-Angabe im Format LSB/MSB im Akku/Y-Register
FLPT	Fließkommazahl im Normalformat
MFLPT	gepacktes Fließkommaformat

Damit das alles nicht so trocken abläuft, soll noch ein kleines Beispiel vorgestellt werden! Die oben schon erwähnte Kotangens-Funktion wird in einem Maschinenprogramm erzeugt, das durch USR anzuspringen ist. In Bild 1 finden Sie ein Flußdiagramm zu dem Programm, welches hier als Hypra-Ass-Listing abgebildet ist (Listing 1). Ein kurzes Testprogramm liefert Listing 2.

Der Einsprung mittels USR bietet den Vorteil, daß der Übergabewert gleich im FLPT-Format im FAC »landet«. Es ist aber sinnvoll, den Übergabeparameter mittels der MOVMF-Routine zu »retten«, weil durch die Kosinus-Funktion der FAC verändert wird. Wenn auch das Ergebnis der Kosinus-Funktion mittels MOVMF beiseite gelegt wurde, holen wir durch MOVFM den Anfangswert wieder in den FAC und bilden mittels SIN den Sinus davon. Schließlich

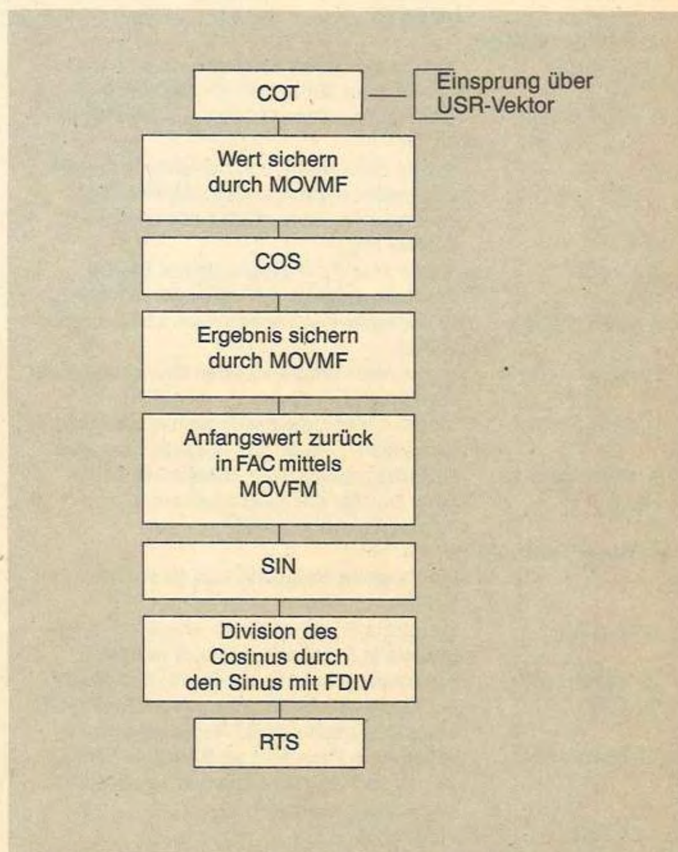


Bild 1. Flußdiagramm einer Kotangens-Funktion

teilen wir den im Speicher stehenden Kosinuswert durch den im FAC befindlichen Sinuswert (unter Verwendung von FDIV). Das Ergebnis ist der Kotangens:

COT X = (COS X/SIN X)

en in eigenen Programmen

Dieser Wert befindet sich nun im FAC und wird mit dem RTS an das Basic-Programm zurückgeliefert. Im Testprogramm weisen wir ihm dann die Variable E zu.

Dieses kurze Beispiel soll Ihnen den Mund wäßrig machen. Sehr viel detaillierter werden die ROM-Routinen im Kurs »Von Basic zu Assembler« behandelt.

(Heimo Ponnath/rs)

Literatur:
1. Kassera/Kassera, Programmieren in Maschinensprache, München 1985: Markt&Technik Verlag, MT 830
2. West, C 64 Computerhandbuch, München 1984, Te-wi
3. Babel/Krause/Dripke, Das Interface Age Systemhandbuch zum C 64, München 1983: Interface Age Verlag
4. Ponnath, C 64 Wunderland der Grafik, München 1985: Markt&Technik Verlag MT 756.

Name	Adresse	Funktion	Eingabe		Ausgabe	
			Format	Orte	Format	Orte
CHRGET	0073	Holt nächstes Byte	1 Byte	Basic-Text	1 Byte	A
CHRGOT	0079	Holt aktuelles Byte	1 Byte	Basic-Text	1 Byte	A
READY	A474	Erzeugt READY-Status	-	-	-	-
LINGET	A96B	Holt Integerwert (0 - 63999)	ASCII-Zahl	Basic-Text	2-Byte-Integer	$14/15
FRMNUM	AD8A	Holt beliebigen numerischen Ausdruck	Basic-Ausdruck	Basic-Text	FLPT	FAC
FRMEVL	AD9E	Holt beliebigen Ausdruck	Basic-Ausdruck	Basic-Text	a)bei Fließkommaz.: FLPT b)bei Integer: FLPT c)bei String: Zeiger auf Deskriptor	 FAC FAC FAC+3/ FAC+4
Diese Routine setzt außerdem eine Reihe von Flaggen: VALTYP ($0D): 0 = Zahl FF = String INTFLAG($0E): 0 = Fließkomma 80 = Integer War Ausdruck einfache Variable, dann zeigt VARNAM ($45/6) das erste Byte des Variablen-Namens						
CHKCLS	AEF7	Prüft auf ») «	ASCII	Basic-Text	-	-
CHKOPN	AEFA	Prüft auf » («	ASCII	Basic-Text	-	-
CHKCOM	AEFD	Prüft auf » , «	ASCII	Basic-Text	-	-
SYNCHR	AEFF	Prüft auf Zeichen im Akkumulator	ASCII	Basic-Text A	-	-
Diese vier Routinen überlesen das Zeichen, wenn vorhanden, ansonsten erfolgt die Ausgabe eines SYNTAX ERROR.						
ISVAR	AF28	Sucht Variablenwert	Name + Kennung	$45/6	a)Zahl: FLPT b)String: Deskriptor	 FAC FAC+3...
ORDVAR	B0E7	Sucht Variablennamen	Name + Kennung	$45/6	Adresse	$47/8
GTBYTC	B79B	Holt Zahl (0 - 255)	ASCII	Basic-Text	1 Byte	X
GETNUM	B7EB	Liest 2 Integerzahlen (Trennung durch Komma) 1.Zahl: 0 - 65535 2.Zahl: 0 - 255	ASCII	Basic-Text	2Byte-Int 1Byte-Int	$14/5 X
COMBYT	E200	Prüft auf » , » und holt folgende Zahl (0 - 255)	ASCII	Basic-Text	1 Byte	X

Tabelle der nützlichen ROM-Routinen
1. Routinen, die die Zusammenarbeit von Basic und Assembler erleichtern.

Name	Adresse	Funktion	Eingabe		Ausgabe	
			Format	Orte	Format	Orte
BLTUC	A3BF	Verschiebt Blöcke	Adressen: Quelle Ziel	Start $5F/60 Ende+1 $5A/5B Ende+1 $58/59		
PUTINT	A9C4	Schiebt FAC als Integer in Variable	FLPT Adresse	FAC $49/50	2Byte-Integer	angegebene Variable
PTFLPT	A9D6	Schiebt FAC in Variable	FLPT Adresse	FAC $49/50	MFLPT	angegebene Variable
GETSPT	AA2C	Schiebt Stringdeskriptor in Variable	Zeiger Adresse	FAC+3 $49/50	Deskript.	angegebene Variable
STRVAL	B7B5	Zahlenstring in FAC einlesen	ASCII Länge	ab $22 A	FLPT	FAC

2.Routinen, die Verschiebungen im Speicher durchführen.

Name	Adresse	Funktion	Eingabe Format	Eingabe Orte	Ausgabe Format	Ausgabe Orte
CONUPK	BA8C	Lädt ARG aus Speicher	MFLPT	A/Y	FLPT	ARG
MOVFM	BBA2	Lädt FAC aus Speicher	MFLPT	A/Y	FLPT	FAC
MOVMF	BBD4	Schiebt FAC in Speicher	FLPT	FAC	MFLPT	angegeb.
			Adresse	X/Y		Speicher
MOVFA	BBFC	ARG in FAC kopieren	FLPT	ARG	FLPT	FAC
MOVAF	BC0C	FAC in ARG kopieren	FLPT	FAC	FLPT	ARG
ACTOFC	BC3C	Akku in FAC schreiben	1 Byte	A	FLPT	FAC

Name	Adresse	Funktion	Eingabe Format	Eingabe Orte	Ausgabe Format	Ausgabe Orte
ASCADD	AA27	Addiert ASCII-Ziffer	ASCII	A	FLPT	FAC
		zu FAC-Inhalt				
OROP	AFE6	FAC = (FAC) OR (ARG)	FLPT	FAC,ARG	FLPT	FAC
ANDOP	AFE9	FAC = (FAC) AND (ARG)	FLPT	FAC,ARG	FLPT	FAC
			0	Y		
FACINX	B1AA	FAC wird als Integer	FLPT	FAC	2Byte-	A/Y
		in A/Y abgelegt			Integer	
UMULT	B357	16-Bit-Multiplikation	2Byte-Integers:		2Byte-	X/Y
			1.Zahl in $28/9		Integer	
			2.Zahl in $71/2			
CIVAYF	B391	Integer(-32768 bis	2Byte-	A/Y	FLPT	FAC
		32767) in FAC	Integer			
SGNFT	B3A2	Integer (0 - 255) in FAC	1 Byte	Y	FLPT	FAC
GETADR	B7F7	Wandelt FAC zu Integer	FLPT	FAC	2Byte-	Y/A
		(0 bis 65535)			Integer	$14/5
FADDH	B849	FAC = FAC + 0,5	FLPT	FAC	FLPT	FAC
FSUB	B850	FAC = Speicherzahl - FAC	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	FAC		
FSUBT	B853	FAC = ARG - FAC	FLPT	FAC,ARG	FLPT	FAC
FADD	B867	FAC = Speicherzahl + FAC	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	FAC		
FADDT	B86A	FAC = ARG + FAC	FLPT	FAC,ARG	FLPT	FAC
COMPLT	B947	Erzeugt 2er-Komplement	FLPT	FAC	FLPT	FAC
		von FAC				
LOG	B9EA	FAC = ln(FAC)	FLPT	FAC	FLPT	FAC
FMULT	BA28	FAC = Speicherzahl * FAC	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	FAC		
FMULTT	BA30	FAC = ARG * FAC	FLPT	FAC,ARG	FLPT	FAC
MUL10	BAE2	FAC = 10 * FAC	FLPT	FAC	FLPT	FAC
DIV10	BAFE	FAC = FAC / 10	FLPT	FAC	FLPT	FACF
DIVF	BB07	FAC = ARG / Speicherzahl	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	ARG		
FDIV	BB0F	FAC = Speicherzahl / FAC	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	FAC		
FDIVT	BB14	FAC = ARG / FAC	FLPT	FAC,ARG	FLPT	FAC
SIGN	BC28	Ermittelt Vorzeichen	FLPT	FAC	1 Byte:	A
		von FAC			1 = +	
					0 = 0	
					FF= -	
ABS	BC58	FAC = ABS(FAC)	FLPT	FAC	FLPT	FAC
FCOMP	BC5B	Vergleicht FAC mit	MFLPT	Zeig.A/Y	1 Byte:	A
		Speicherzahl	FLPT	FAC	FAC > Sp. = 1	
					FAC = Sp. = 0	
					FAC < Sp. = FF	
INT	BCCC	FAC = INT(FAC)	FLPT	FAC	FLPT	FAC

3.Routinen zur Arithmetik

Name	Adresse	Funktion	Eingabe Format	Eingabe Orte	Ausgabe Format	Ausgabe Orte
AADD	BD7E	Addiert A zu FAC	FLPT	FAC	FLPT	FAC
			1 Byte	A		
SQR	BF71	FAC = SQR(FAC)	FLPT	FAC	FLPT	FAC
MPOT	BF78	FAC = Speicherzahl) FAC	MFLPT	Zeig.A/Y	FLPT	FAC
			FLPT	FAC		
FPWRT	BF7B	FAC = ARG) FAC	FLPT	FAC,ARG	FLPT	FAC
NEGOP	BFB4	FAC = -FAC	FLPT	FAC	FLPT	FAC
EXP	BFED	FAC = e) FAC	FLPT	FAC	FLPT	FAC
POLYX	E059	Polynomberechnung	Adresse	Zeig.A/Y	FLPT	FAC
		FAC = $a_0 + a_1x + a_2x^2 + \dots$				
	Der Zeiger A/Y weist auf den Start der Konstantentabelle mit dem Aufbau:					

		1.Byte = Polynomgrad Die weiteren Bytes sind die Koeffizienten des Polynoms in der Reihenfolge $a_n, \ldots, a_0$ im MFLPT-Format.				
COS	E264	FAC = COS(FAC)	FLPT	FAC	FLPT	FAC
SIN	E26B	FAC = SIN(FAC)	FLPT	FAC	FLPT	FAC
TAN	E2B4	FAC = TAN(FAC)	FLPT	FAC	FLPT	FAC
ATN	E30E	FAC = ATN(FAC)	FLPT	FAC	FLPT	FAC

3a. Weitere Routinen zur Arithmetik.

Name	Adresse	Funktion	Eingabe		Ausgabe	
			Format	Orte	Format	Orte
ERROR	A437	Fehlermeldung ausgeben und READY	Fehler-nummer	X	ASCII	Bildschirm
LIST	A69C	Listet Basicprogramm	-	-	-	-
NUMDON	AABC	Druckt FAC auf Bildschirm aus	FLPT	FAC	ASCII	Bildschirm
STROUT	AB1E	Gibt String a. Bildschirm aus (Ende = 0)	Adresse	Zeig.A/Y	ASCII	Bildschirm
SYNERR	AF08	Ausgabe SYNTAX ERROR	-	-	ASCII	Bildschirm
OVERR	B97E	Ausgabe OVERFLOW ERROR	-	-	ASCII	Bildschirm
LINPRT	BDCD	Druckt Integerzahl aus (0 - 65535)	2Byte-Integer	X/A	ASCII	Bildschirm
FACOUT	BDD7	Druckt FAC auf Bild-schirm aus	FLPT	FAC	ASCII	Bildschirm
FOUT	BDDD	FAC wird zu ASCII-String mit Endekennung 0 Kann direkt mit STROUT ausgegeben werden.	FLPT	FAC	ASCII Startadr.	ab $100 A/Y
SAVET	E156	Save	Parameter aus dem Basic-Text			
VERFYT	E165	Verify	Parameter aus dem Basic-Text			
LOADT	E168	Load	Parameter aus dem Basic-Text			
SLPARA	E1D4	Holt Parameter für Save, Load, Verify aus dem Basic-Text				
PLOTK	E50A	Setzt Cursorposition	Zeile Spalte	X Y		
HOME	E566	Cursor in Home-Position				
PLOTR	E56C	Setzt Cursorposition	Zeile Spalte	$D6 $D3		
GETKBC	E5B4	Holt Zeichen aus Tastaturpuffer	-	-	1 Byte	A
PRT	E716	Gibt Zeichen in Akku auf Bildschirm aus	1 Byte	A	ASCII	Bildschirm
CLRLN	E9FF	Löscht Xte Bildschirm-zeile	Zeilennr.	X	-	-

4.Auswahl von Ein- und Ausgabe-Routinen

```
hypra-ass  assemblerlisting:

             10   -          .li 1,4,7
             20   -          .ba $6000
;
;einsprung mittels usr
;zuvor usr-vektor einstellen!
;
             160  -          .eq cos=$e264
             165  -          .eq movfm=$bba2
             170  -          .eq movmf=$bbd4
             180  -          .eq sin=$e26b
             190  -          .eq fdiv=$bb0f
             200  -          .eq wert=$7000
             205  -          .eq wert1=$7010
;
6000 a210    :212  -start    ldx #<(wert1)
6002 a070    :214  -         ldy #>(wert1)
6004 20d4bb  :216  -         jsr movmf
6007 2064e2  :220  -         jsr cos
600a a200    :230  -         ldx #<(wert)
600c a070    :240  -         ldy #>(wert)
600e 20d4bb  :250  -         jsr movmf
6011 a910    :252  -         lda #<(wert1)
6013 a070    :254  -         ldy #>(wert1)
6015 20a2bb  :256  -         jsr movfm
6018 206be2  :260  -         jsr sin
601b a900    :270  -         lda #<(wert)
601d a070    :280  -         ldy #>(wert)
601f 200fbb  :290  -         jsr fdiv
6022 60      :300  -         rts
;
             320  -          .sy 1,4,7

symbols in alphabetical order:

cos          = $e264
fdiv         = $bb0f
movfm        = $bba2
movmf        = $bbd4
sin          = $e26b
start        = $6000
wert         = $7000
wert1        = $7010

end of assembly  0:25.9
base = $6000  last byte at $6022
```

Listing 1. Hypra-Ass-Listing der Kotangens-Funktion

```
10 REM***TEST FUER COTANGENS***
20 POKE785,0:POKE786,96:REM USR-VEKTOR
30 INPUT"WINKEL";W:W=W*π/180:REM AUF BOGENMASS
40 E=USR(W):REM AUFRUF DES PROGRAMMES
50 PRINTW,E:REM ERGEBNIS IN E
60 END
READY.
```

Listing 2. Test der Kotangens-Funktion

Fortsetzung von Seite 155

```
c830 : 27 45 52 2c 20 28 43 29  b7
c838 : 20 31 39 38 35 20 42 59  56
c840 : 20 4d 2e 20 57 45 48 4e  f4
c848 : 45 52 20 2e 2e 2e 41 41  60
c850 : 41 42 42 42 52 42 42 43  52
c858 : 43 53 43 43 45 49 49 50  e2
c860 : 4c 44 42 44 43 4e 43 44  24
c868 : 4c 53 54 42 54 4f 53 42  4d
c870 : 49 50 4a 4c 4c 50 50 53  2d
c878 : 42 43 53 52 53 53 54 54  45
c880 : 54 54 3f 42 4a 52 52 45  22
c888 : 42 42 44 4e 53 43 43 45  ad
c890 : 4f 50 56 4c 45 4d 50     eb
c898 : 4f 4e 4e 4c 44 45 56 45  7e
c8a0 : 4c 4f 50 45 44 42 59 4d  a7
c8a8 : 41 52 54 49 4e 4c 53 44  6e
c8b0 : 53 48 48 45 4e 4c 45 4f  dd
c8b8 : 54 54 41 53 58 58 3f 52  db
c8c0 : 4d 54 54 51 41 59 43 44  eb
c8c8 : 4c 43 53 51 52 4c 53 43  10
c8d0 : 49 43 50 58 52 43 59 50  1f
c8d8 : 41 43 43 58 44 50 59 59  76
c8e0 : 58 43 41 49 58 41 59 54  f1
c8e8 : 58 41 52 59 52 41 50 44  9a
c8f0 : 45 56 49 4c 41 58 59 58  29
c8f8 : 41 53 3f 4b 50 49 53 38  29
c900 : 22 02 02 02 22 05 02 2b  8d
c908 : 22 05 02 02 22 05 02 0a  d4
c910 : 22 02 02 02 22 05 02 0c  5f
c918 : 22 02 02 02 22 05 02 27  9d
c920 : 04 02 02 24 04 30 02 14  1c
c928 : 04 30 02 24 04 30 02 20  54
c930 : 04 02 02 02 04 30 02 0e  dc
c938 : 04 02 02 02 04 30 02 3a  3c
c940 : 11 02 02 02 11 29 02 2a  ca
c948 : 11 29 02 39 11 29 02 17  26
c950 : 11 02 02 02 11 29 02 0d  a0
c958 : 11 02 02 02 11 29 02 3b  04
c960 : 03 02 02 02 03 09 02 26  f2
c968 : 03 09 02 39 03 09 02 0b  2e
c970 : 03 02 02 02 03 09 02 2f  14
c978 : 03 02 02 02 03 09 02 02  c2
c980 : 31 02 02 23 31 32 02 1c  7c
c988 : 02 35 02 23 31 32 02 06  c3
c990 : 31 02 02 23 31 32 02 1f  92
c998 : 31 36 02 02 31 02 02 28  21
c9a0 : 15 1d 02 28 15 1d 02 33  72
c9a8 : 15 21 02 28 15 1d 02 07  24
c9b0 : 15 02 02 28 15 1d 02 2e  eb
c9b8 : 15 34 02 28 15 1d 02 1b  e5
c9c0 : 0f 02 02 1b 0f 16 02 13  84
c9c8 : 0f 18 02 1b 0f 16 02 2d  cb
c9d0 : 0f 02 02 02 0f 16 02 19  7d
c9d8 : 0f 02 02 02 0f 16 02 10  73
c9e0 : 1e 02 02 10 1e 12 02 25  47
c9e8 : 1e 1a 02 10 1e 12 02 08  21
c9f0 : 1e 02 02 02 1e 12 02 2c  a3
c9f8 : 1e 02 02 02 1e 12 02 00  53
ca00 : 34 80 80 80 04 04 80 00  07
ca08 : 08 00 80 80 06 06 80 01  d5
ca10 : 54 80 80 80 24 24 80 00  3a
ca18 : 42 80 80 80 26 26 80 02  64
ca20 : 34 80 80 04 04 04 80 00  97
ca28 : 08 00 80 06 06 06 80 01  a6
ca30 : 54 80 80 80 24 24 80 00  5a
ca38 : 42 80 80 80 26 26 80 00  80
ca40 : 34 80 80 80 04 04 80 00  47
ca48 : 08 00 80 02 06 06 80 01  45
ca50 : 54 80 80 80 24 24 80 00  7a
ca58 : 42 80 80 80 26 26 80 00  a0
ca60 : 34 80 80 80 04 04 80 00  67
ca68 : 08 00 80 12 06 06 80 01  67
ca70 : 54 80 80 80 24 24 80 00  9a
ca78 : 42 80 80 80 26 26 80 80  c1
ca80 : 34 80 80 04 04 04 80 00  f7
ca88 : 80 00 80 06 06 06 80 01  7e
ca90 : 54 80 80 24 24 44 80 00  2f
ca98 : 42 00 80 80 26 80 80 08  83
caa0 : 34 08 80 04 04 04 80 00  db
caa8 : 08 00 80 06 06 06 80 01  26
cab0 : 54 80 80 24 24 44 80 00  4f
cab8 : 42 00 80 26 26 46 80 08  86
cac0 : 34 80 80 04 04 04 80 00  37
cac8 : 08 00 80 06 06 06 80 01  46
cad0 : 54 80 80 80 24 24 80 00  fa
cad8 : 42 80 80 80 26 26 80 08  30
cae0 : 34 80 80 04 04 04 80 00  57
cae8 : 08 00 80 06 06 06 80 01  66
caf0 : 54 80 80 80 24 24 80 00  1a
caf8 : 42 80 80 80 26 26 80 00  40
cb00 : ac 00 f7 00 ff 00 ff 00  aa
```

Listing 1. »REASS« (Schluß)

Impressum

Herausgeber: *Carl-Franz von Quadt, Otmar Weber*

Chefredakteur: *Albert Absmeier*
Stellv. Chefredakteur: *Georg Klinge*
Leitender Redakteur: *Gottfried Knechtel – verantwortlich für den redaktionellen Teil*
Redaktion: *Klaus Schrödl, Ralf Sablowski*
Redaktionsassistenz: *Andrea Kaltenhauser, Brigitte Bobenstetter, Helga Weber (202)*
Hotline: *Monika Welzel (640)*
Mitarbeiter der Redaktion: *Heimo Ponnath*

Alle Artikel sind mit dem Kennzeichen des Redakteurs (kn = Gottfried Knechtel, sk = Klaus Schrödl, rs = Ralf Sablowski) und/oder mit dem Namen des Autors/Mitarbeiters gekennzeichnet

Art-director: *Friedemann Porscha*
Layout: *Erich Schulze (Cheflayouter), Marian Schwarz*
Fotografie: *Jens Jancke, Sabine Tennstaedt,*
Titelgestaltung: *Friedemann Porscha,*
Spritzgrafik: *Ewald Standke*
Computergrafik: *Werner Nienstedt*

Auslandsrepräsentation:
Schweiz: *Markt&Technik Vertriebs AG, Kollerstr. 3, CH-6300 Zug, Tel. 042-415656, Telex: 862329 mut ch*
USA: *M&T Publishing Inc.; 501 Galveston Drive Redwood City, CA 94063, Telefon: (415) 366-3600, Telex 752-351*
Österreich: *Markt&Technik Ges. mbH Hermann Raniger, Große Neugasse 28, A 1040-Wien, Tel. 0043-222-8579455, Telex: 047-132532*

Manuskripteinsendungen: *Manuskripte und Programmlistings werden gerne von der Redaktion angenommen. Sie müssen frei sein von Rechten Dritter. Sollten sie auch an anderer Stelle zur Veröffentlichung oder gewerblichen Nutzung angeboten worden sein, muß dies angegeben werden. Mit der Einsendung von Manuskripten und Listings gibt der Verfasser die Zustimmung zum Abdruck in von der Markt&Technik Verlag AG herausgegebenen Publikationen und zur Vervielfältigung der Programmlistings auf Datenträger. Mit der Einsendung von Bauanleitungen gibt der Einsender die Zustimmung zum Abdruck in von Markt&Technik Verlag AG verlegten Publikationen und dazu, daß Markt&Technik Verlag AG Geräte und Bauteile nach der Bauanleitung herstellen läßt und vertreibt oder durch Dritte vertreiben läßt. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte und Listings wird keine Haftung übernommen.*

Produktionsleiter: *Klaus Buck (180)*

Anzeigenverkaufsleitung: *»Populäre Computerzeitschriften«: Alexander Narings (780)*
Anzeigenleitung: *Phillip Schiede (399) – verantwortlich für Anzeigen*

Anzeigenformate: *½-Seite ist 266 Millimeter hoch und 185 Millimeter breit (2 Spalten à 86 Millimeter oder 4 Spalten à 43 Millimeter). Vollformat 297x210 Millimeter.*

Anzeigenpreise: *Es gilt die Anzeigenpreisliste vom 5. Januar 1988. ½-Seite sw: DM 5400,-. Farbzuschlag: erste und zweite Farbe aus der Europa-Skala je DM 1000,-. Vierfarbzuschlag DM 2800,-. Plazierung innerhalb der redaktionellen Beiträge. Mindestgröße ¼-Seite.*

Anzeigenverwaltung und Disposition: *Lisa Landthaler (233)*

Anzeigen-Auslandsvertretung: England: *F. A. Smyth & Associates Limited, 23a, Aylmer Parade, London, N2 OPQ. Telefon: 0044/1/3405058, Telefax: 0044/1/3419602*
Taiwan: *Third Wave Publishing Corp., 1-4 Fl. 977 Min Shen E. Road, Taipei 10581, Taiwan, R.O.C., Tel. 00886/2/7630052, Telefax: 00886/2/7658767, Telex: 078529335*

Vertriebsleiter: *Helmut Grünfeldt (189)*

Leiter Vertriebs-Marketing: *Benno Gaab (740)*

Vertrieb Handelsauflage: *Inland (Groß-, Einzel- und Bahnhofsbuchhandel) sowie Österreich und Schweiz: Pegasus Buch- und Zeitschriften-Vertriebs GmbH, Hauptstätter Straße 96, 7000 Stuttgart 1,*

Bezugsmöglichkeiten: *Leser-Service: Telefon (089) 4613-249. Bestellungen nimmt der Verlag oder jede Buchhandlung entgegen.*

Preis: *Das Einzelheft kostet DM 14,–*

Druck: *SOV Graphische Betriebe, Laubanger 23, 8600 Bamberg*

Redaktionsdirektor: *Michael M. Pauly*

Vorstand: *Otmar Weber (Vors.), Bernd Balzer, Werner Brodt*

Leiter Unternehmensbereich »Populäre Computerzeitschriften«: *Michael Scharfenberger*

Redaktionskoordination »Populäre Computerzeitschriften«: *Hans-Günther Beer*

Anschrift für Verlag, Redaktion, Vertrieb, Anzeigenverwaltung und alle Verantwortlichen: *Markt&Technik Verlag Aktiengesellschaft, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0, Telex 5-22052*

ISSN 0931-8933

Telefon-Durchwahl im Verlag: *Wählen Sie direkt: Per Durchwahl erreichen Sie alle Abteilungen direkt. Sie wählen 089/4613 und dann die Nummer, die in den Klammern hinter dem jeweiligen Namen angegeben ist.*



